

«Серия "Недостающее руководство" — просто самая разумная и полезная серия руководств».
КЕВИН КЕЛЛИ, СОУЧРЕДИТЕЛЬ ЖУРНАЛА «WIRED»

HTML5

недостающее руководство

Книга, которая должна быть на полке



O'REILLY®

Мэтью Мак-Дональд

HTML5

the missing manual®

The book that should have been in the box®

Matthew MacDonald

O'REILLY®

Beijing | Cambridge | Farnham | Köln | Sebastopol | Tokyo

Мэтью Мак-Дональд

HTML5

недостающее руководство

Санкт-Петербург
«БХВ-Петербург»
2012

УДК 681.3.068
ББК 32.973.26-018.1
М15

Мак-Доналд М.

М15 HTML5. Недостающее руководство: Пер. с англ. — СПб.: БХВ-Петербург, 2012. — 480 с.: ил.

ISBN 978-5-9775-0805-6

Доступно и в занимательной форме рассказано, как HTML превратился в HTML5. Рассмотрены семантические элементы и новые стандарты языка. Описано, как создавать современные веб-страницы, в том числе улучшенные веб-формы, поддерживать аудио и видео, рисовать на холсте, совершенствовать оформление веб-страниц с помощью CSS3. Даны практические рекомендации по созданию интеллектуальных веб-приложений, хранению данных, разработке автономных приложений. Показано, как реализовать взаимодействие с веб-сервером, геолокацию, фоновые вычисления, управление историей просмотров и браузерную совместимость с элементами HTML5.

Для веб-разработчиков

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Елена Васильева*
Перевод с английского	*Сергея Таранушенко*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Оформление обложки	*Марины Дамбиевой*

Подписано в печать 30.04.12.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 38,7.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-1-449-30239-9 (англ.)
ISBN 978-5-9775-0805-6 (рус.)

Оглавление

Об авторе



Мэтью Мак-Дональд (Matthew MacDonald) — автор научных и технических книг, из-под пера которого вышло свыше десятка книг. Начинающие веб-разработчики могут войти в интернет-сообщество с помощью его книги "Создание веб-сайта. Основное руководство"[1] (Creating a Website: The Missing Manual). Офисные трудяги могут щелкать числа, как орешки, с книгой "Excel 2007. Недостающее руководство"[2] (Excel 2010: The Missing Manual). А люди разумные всех сфер деятельности могут сделать для себя открытие, насколько странными они являются, с помощью книг "Ваш мозг. Недостающее руководство" (Your Brain: The Missing Manual) и "Ваше тело. Недостающее руководство" (Your Body: The Missing Manual).

Благодарности

Никакому автору не было бы под силу написать книгу без небольшой армии помощников. Я в неоплатном долгу перед всей командой проекта "Missing Manual", особенно перед моим редактором Нэн Барбер, которая никогда не терялась в ползучих песках HTML5, и перед профессиональными техническими рецензентами Шелли Пауэрс и Стивом Сюэрингом, которые помогали выявлять вкравшиеся ошибки и давали неизменно хорошие советы. И, как всегда, я признателен многим помощникам, которые усиленно работали, индексируя страницы, создавая рисунки и вычитывая окончательный вариант текста.

Наконец, за моменты моей жизни, проведенные вне этой книги, я бы хотел сказать спасибо всем членам моей семьи. Это мои родители, Нора и Поль, мои крёстные родители, Разя и Хамид (Razia и Hamid), моя жена Фариа (Faria) и мои дочери Мая и Бренна (Maya и Brenna). Спасибо вам всем!

Мэтью Мак-Дональд

---

[1] Мак-Дональд М. Создание Web-сайтов. Основное руководство. — М.: Эксмо, 2010.

[2] Мак-Дональд М. Excel 2007. Недостающее руководство. — СПб.: БХВ-Петербург, Русская редакция, 2008.

Введение

С первого взгляда можно предположить, что HTML5 — это пятая версия языка HTML для создания веб-страниц. Но в действительности все не так просто.

HTML5 — неформал. Его придумала группа вольнодумцев, которые не входили в группу, отвечавшую за официальный стандарт HTML. В стандарте HTML5 разрешаются методы написания страниц, которые были запрещены десять лет тому назад. В нем подробно изложены инструкции браузерам, как обрабатывать ошибки в разметке страниц, чтобы попытаться отобразить эти страницы, вместо того чтобы сразу же забраковывать их. Он, наконец, позволяет воспроизведение видео, не прибегая к помощи модулей расширения браузера, таких как, например, Flash. Также в этом стандарте вводится лавина функциональностей, движимых JavaScript, которые могут придать веб-страницам определенные расширенные, интерактивные возможности, встречаемые в программном обеспечении для настольных компьютеров.

Разобраться в HTML5 — задача не из легких. Самой большой проблемой является то обстоятельство, что термин "HTML5" используется для обозначения свыше дюжины отдельных стандартов. (Как мы узнаем, эта ситуация является результатом эволюции HTML5, который начинался как единый стандарт, но впоследствии был разделен на более управляемые части.) В действительности, термин "HTML5" сейчас означает "HTML5 и связанные стандарты" и, в еще более широком понимании, "следующее поколение технологий разработки веб-страниц". Вот такую версию HTML5 мы и будем рассматривать в этой книге: все от базового языка HTML5 до новых возможностей, сброшенных в одну кучу с HTML5, хотя они *никогда* и не были частью этого стандарта.

Это приводит нас ко второй потенциальной проблеме с HTML5 — браузерной поддержке. Разные браузеры поддерживают различные части HTML5; кроме этого, есть такие возможности, которые не работают ни в одном из браузеров.

Несмотря на все эти трудности, никто не ставит под сомнение, что HTML5 — это будущее веб-дизайна. Он поддерживается крупными компаниями — разработчиками программного обеспечения, такими как Apple и Google; организация W3C (World Wide Web Consortium, Консорциум Всемирной паутины) прекратила свою работу над стандартом XHTML, чтобы формализовать и поддерживать стандарт HTML5; кроме этого, все разработчики браузеров поддерживают значительную

часть этого стандарта. Прочитав эту книгу и усвоив ее материал, вы также сможете стать членом клуба веб-новаторов и создавать искусные веб-страницы, подобные показанной на рис. В1.

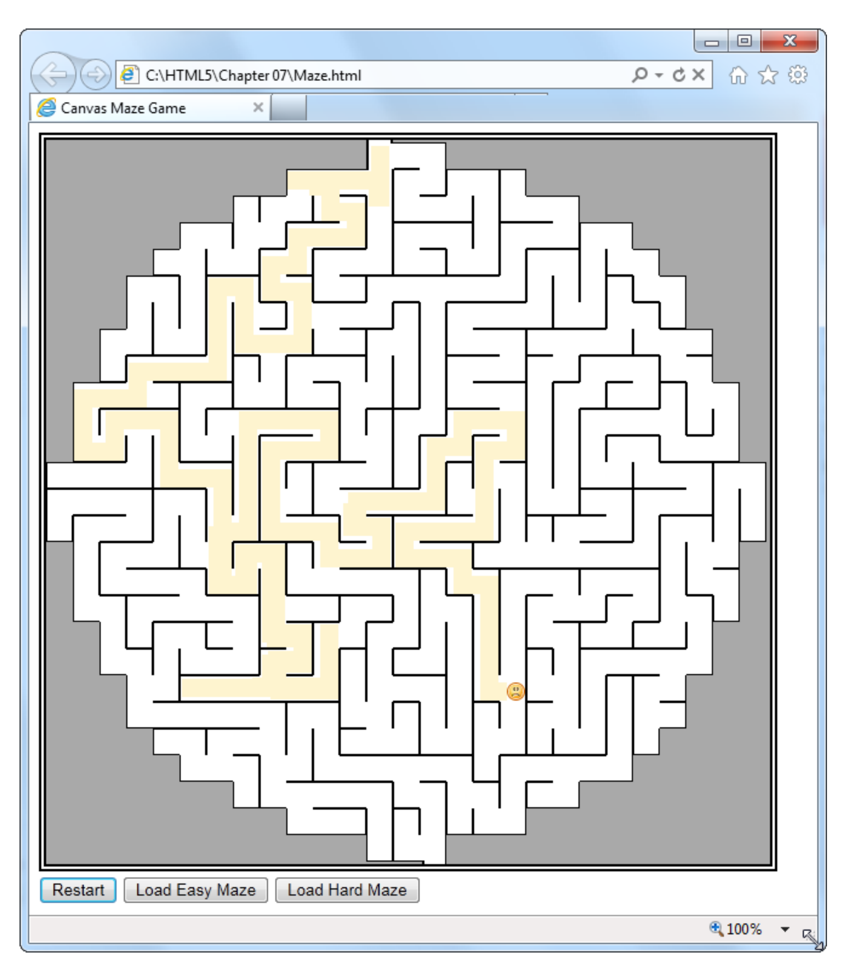


Рис. В1. В темные времена Интернета (иными словами, где-то около года тому назад) для создания игр на веб-страницах нужно было обращаться к помощи модулей расширения браузера, таких как, например, Flash. Но благодаря новым возможностям HTML5, включая холст (применение которого показано на этом рисунке), теперь это можно делать с помощью надежного языка JavaScript без модулей расширения. На этом рисунке HTML5 приводит в действие игру "Лабиринт", которая рассматривается в *главе 7*

Что нужно знать для работы с этой книгой?

В этой книге рассматривается HTML5 — самая последняя и самая мощная версия стандарта HTML. И хотя вам не обязательно быть профессионалом по разметке, нужно все-таки иметь определенные навыки в области веб-разработки. Далее приведено описание основных необходимых навыков.

- **Создание веб-страниц.** Предполагается, что у вас есть опыт создания хотя бы нескольких веб-страниц или, по крайней мере, вы понимаете, как использовать элементы HTML для структурирования содержимого в заголовки, абзацы и списки. Если вы полностью несведущи в области веб-разработки, вам лучше начать с чего-то более легкого.
- **Опыт работы с таблицами стилей.** Современный веб-сайт невозможен без применения технологии CSS (Cascading Style Sheet, каскадные таблицы стилей), посредством которой веб-страницы компонуются и форматируются. Чтобы понимать излагаемый в этой книге материал, вам нужно знать основы таблиц стилей — как создавать их, что помещается внутрь их, а также как подключать их к веб-странице. Если ваши знания по этой теме несколько туманны, можете почерпнуть основные сведения в *приложении 1*. Но если материала из *приложения 1* вам недостаточно или же вы хотите повысить уровень своих знаний в этой области до степени, позволяющей создавать по настоящему искусные компоновки и стили, воспользуйтесь какой-либо дополнительной книгой.
- **Опыт работы с JavaScript.** Нет, для создания веб-страниц язык JavaScript не требуется. Но если вы хотите использовать многие ловкие возможности HTML5, такие как рисование на холсте или взаимодействие с веб-сервером, без JavaScript вам не обойтись. Если у вас имеется хоть какой-либо опыт программирования, но вы не знакомы с JavaScript, тогда материал в *приложении 2* поможет вам чуть-чуть разбираться в этом языке. Но если ваши познания программирования нулевые, тогда вы не сможете должным образом понять многие рассматриваемые в этой книге примеры. В таком случае вам следует ознакомиться с более подробным введением в программирование вообще и в программирование на JavaScript в частности.

Эти требования, к сожалению, не обойти стороной. Такова стоимость возможности работать с передовыми технологиями веб-дизайна.

Написание кода HTML5

Разметку HTML5 можно создавать с помощью тех же приложений, что и разметку обычного HTML. Эти приложения могут быть такими простыми, как базовый текстовый редактор Блокнот (для платформ Windows) или TextEdit (для платформ Mac). Многие современные средства разработки (такие как Adobe Dreamweaver или Microsoft Expression Web) имеют шаблоны, посредством которых можно быстро создавать новые документы HTML5. Но базовая структура страницы HTML5 настолько простая, что ее можно создать с помощью любого веб-редактора, даже если этот веб-редактор не предназначен для работы именно с HTML5.

> ***Примечание***
>
> И, конечно же, не имеет никакого значения, на какой платформе создавать и просматривать веб-страницы, будь то платформа Windows или самая последняя версия Mac OS X — HTML5 поддерживает операционные системы всех типов.

Просмотр страницы HTML5

В отношении HTML5 часто задается вопрос: какие браузеры поддерживают этот стандарт? К сожалению, на этот вопрос нет четкого ответа. Как мы увидим в этой книге, HTML5 в действительности является коллекцией независимых стандартов. Некоторые его части уже поддерживаются, а некоторые не будут поддерживаться еще в течение нескольких лет (а какие-то, скорее всего, вообще не будут поддерживаться никогда). Ситуация с остальными частями пока не ясна. Это означает, что HTML5 работает на некоторых версиях некоторых браузеров.

Вот список основных браузеров, которые поддерживают значительную часть возможностей HTML5, не требуя резервных решений:

- Internet Explorer 9 и более поздние версии;
- Firefox 3.5 и более поздние версии;
- Google Chrome 8 и более поздние версии;
- Safari 4 и более поздние версии;
- Opera 10.5 и более поздние версии.

Уровень поддержки браузерами HTML5 возрастает с увеличением версии браузера. Например, уровень поддержки HTML5 в Firefox 5 выше, чем в Firefox 3.5.

До рекомендации применять новую возможность HTML5 в книге ясно указывается текущий уровень поддержки этой возможности основными браузерами. Но разработчики браузеров выпускают новые версии сравнительно часто, поэтому вам следует проводить собственные исследования по текущей поддержке, прежде чем применять в своей странице возможность, которая способна вызвать проблемы. В этом отношении можно воспользоваться услугами веб-сайта **http://caniuse.com**, который предоставляет точную информацию о поддержке определенных возможностей конкретными браузерами. (Это полезное средство рассматривается более подробно в *разд. "Поддерживает ли браузер вашу разметку?" главы 1*.)

Примечание

В этой книге рассматриваются некоторые возможности, о которых заведомо известно, что они не работают в некоторых браузерах. Не впадайте по этому поводу в панику. Это вполне приемлемо, если вы хотите только получить представление обо всем диапазоне возможностей HTML5, но при этом фокусироваться на тех из них, которые можно использовать уже сейчас. А "сырые" функциональности можно рассматривать как демонстрацию того, что нас ожидает в Интернете в будущем.

Когда HTML5 будет готов?

Общий ответ на этот вопрос таков: он уже готов сейчас. Даже постылый Internet Explorer 6, которому 10 лет и который битком набит своеобразными функциями обработки веб-страниц, часто должным образом не отображающими эти страницы, может отображать элементы HTML5. Это достигается тем, что стандарт HTML5 преднамеренно разрабатывался так, чтобы включать и расширять традиционный HTML.

Как мы уже узнали, HTML5 в действительности является коллекцией разных стандартов с разным уровнем поддержки браузерами. Поэтому, хотя любой веб-разработчик может переключиться на использование документов HTML5 уже сегодня (и некоторые крупные сайты, такие как Google, YouTube и Wikipedia, уже перешли), может пройти несколько лет, прежде чем использование большей части новых хитроумных возможностей HTML5 станет безопасным. По крайней мере, без добавления какого-либо механизма резервных решений для менее сообразительных браузеров.

Примечание

По большому счету не имеет особого значения, к какой спецификации принадлежит та или иная возможность. Важна современная поддержка браузерами этой возможности, а также вероятность ее поддержки в будущем. Когда в этой книге рассматривается какая-либо новая возможность, указывается, в какой спецификации она определяется, и текущий уровень ее поддержки основными браузерами.

Разработчикам, которые трепетно относятся к стандартам, может быть интересно знать, насколько близки различные стандарты к моменту объявления официального статуса. Дать точный ответ на этот вопрос сложно, поскольку создатели HTML5 считают более важным то, что поддерживается браузерами, нежели то, что указано в стандарте. Иными словами, веб-разработчики могут использовать по желанию любые возможности уже сейчас, если могут заставить их работать. Но веб-разработчики, крупные компании, государственные органы и другие организации часто составляют свое мнение о готовности языка к применению по официальному статусу его стандарта.

Вообще, в настоящее время спецификация HTML5 находится в руках организации W3C в качестве *рабочего проекта* (working draft). Данное определение обозначает, что это вполне установившийся стандарт, но такой, что еще может измениться в процессе прохождения этапа *возможной рекомендации* (candidate recommendation), в который он войдет где-то в 2012 г. А вхождение в стадию *рекомендации* (recommendation) может занять многие годы, т. к. для этого стандарту требуется пройти многочисленные тестирования. Но это уже не так важно, т. к. на этом этапе изменения будут немногочисленные, и все, кто хочет использовать HTML5, уже сядут на этот поезд.

О чем эта книга?

Двенадцать глав этой книги содержат всеохватывающее руководство по HTML5.

Часть I. Знакомство с новым языком.

- В *главе 1 "Представляем HTML5"* рассматривается, как HTML превратился в HTML5. Мы познакомимся с первым документом HTML5, увидим, какие изменения претерпел язык, а также обсудим поддержку возможностей браузерами.
- В *главе 2 "Новый способ структурирования страниц"* рассматриваются семантические элементы HTML5 — группа элементов, которые могут придать смысл

разметке. При правильном использовании эта дополнительная информация может оказаться полезной для браузеров, средств чтения экрана, инструментов для веб-дизайна и поисковых систем.

- В *главе 3 "Разметка со смыслом"* обсуждается семантика и рассматриваются дополнительные стандарты, такие как *микроданные*. И хотя эти сведения могут показаться несколько теоретизированными, веб-разработчиков, которые разберутся в них, ожидает достойное вознаграждение — лучшие, более подробные результаты поисковых запросов в поисковых системам, таких как Google.

Часть II. Создание современных веб-страниц.

- В *главе 4 "Продвинутые веб-формы"* исследуются изменения, внесенные HTML5 в элементы веб-форм — текстовые поля, списки, флажки и прочие элементы управления, используемые для сбора информации от посетителей страницы. HTML5 добавляет несколько новых примочек и некоторые базовые средства для улавливания ошибок ввода.
- В *главе 5 "Аудио и видео"* изучается одна из самых захватывающих возможностей HTML5 — поддержка воспроизведения видео и аудио. Мы научимся выживанию в битве видеокодеков для веба, чтобы создавать страницы, воспроизводящие мультимедиа во всех браузерах, и даже увидим, как создать собственный проигрыватель.
- В *главе 6 "Основы рисования на холсте"* рассматривается двумерная поверхность для рисования — холст. На этом холсте мы научимся рисовать фигуры, изображения и текст, и даже на его основе создадим простую программу рисования (применив JavaScript).
- В *главе 7 "Продвинутые методы работы с холстом"* мы разовьем наши навыки работы с холстом и узнаем, как создавать тени и вычурные узоры, а также освоим более продвинутые методы, такие как создание анимации и фигур, активизируемых щелчком мыши.
- В *главе 8 "Совершенствование стилей с помощью CSS3"* рассказывается о последней версии стандарта CSS, которая хорошо дополняет HTML5. Мы научимся украшать текст вычурными шрифтами, настраивать страницу для отображения на разных типах мобильных устройств и добавлять бросающиеся в глаза эффекты с помощью переходов.

Часть III. Создание интеллектуальных веб-приложений.

- В *главе 9 "Хранение данных"* рассматривается новая функциональность — веб-хранилище, которое позволяет странице сохранять информацию на компьютере посетителя. (Оно похоже на cookies, но на порядок эффективнее.) Мы также научимся обрабатывать выбранные пользователем файлы посредством JavaScript-кода прямо в веб-странице, а не отправляя их для этого на сервер.
- В *главе 10 "Автономные приложения"* исследуется новая функциональность кэширования, которая позволяет браузеру работать с веб-страницей даже при отсутствии подключения к Интернету.

- В *главе 11 "Взаимодействие с веб-сервером"* мы познакомимся со сложным обменом информацией с веб-сервером. Мы начнем с изучения испытанного временем объекта `XMLHttpRequest`, который позволяет посредством JavaScript отправлять веб-серверу запросы информации. После этого мы перейдем к рассмотрению двух новых возможностей: отправляемых сервером событий и более амбициозной, но не совсем доведенной, функциональности веб-сокетов.
- В *главе 12 "Несколько полезных возможностей на JavaScript"* рассматриваются три дополнительные возможности, направленные на решение задач современного Интернета. Сначала мы увидим, как использовать геолокацию, чтобы установить местоположения посетителя страницы. Потом мы воспользуемся веб-работниками для выполнения трудоемких задач в фоновом режиме. Наконец, мы исследуем функциональность отслеживания истории просмотров страниц, которая позволяет синхронизировать URL динамически обновляемой страницы с ее текущим содержимым.

Кроме этого, два приложения помогут вам получить базовые знания, необходимые для овладения HTML5. В *приложении 1* дается краткий обзор CSS, а в *приложении 2* — JavaScript.

Онлайновые ресурсы

Купив эту книгу, вы приобрели больше, чем просто книгу для чтения. Вместе с ней вы получили доступ к онлайновым файлам примеров, а также советам, статьям и паре-другой видео.

Недостающий CD

Эта книга не имеет сопровождающего компакт-диска, но это не значит, что вы что-либо теряете. Все рассматриваемые в этой книге примеры можно загрузить по адресу **http://missingmanuals.com/cds/html5mm**. Кроме этого, чтобы вам не стирать пальцы, вводя URL упоминаемых в книге веб-сайтов, все они предоставлены в виде ссылок на указанной веб-странице.

> **Совет**
>
> Если вы ищете конкретный пример, самый быстрый способ найти его — посмотреть на соответствующий рисунок в книге. В конце строки адреса обычно будет название файла. Например, на рис. 1.1 указывается путь c:\HTML5\Chapter01\SuperSimpleHTML5.html, что говорит нам, что файл примера называется SuperSimpleHTML5.html.

Сайт для тестирования примеров книги

Рассматриваемые в книге примеры можно испытать еще одним способом — на странице по адресу **http://www.prosetech.com/html5/**. Эта страница содержит все примеры из данной книги, которые можно открыть в браузере. Просмотр примеров с этой страницы может помочь вам избежать многих проблем, т. к. HTML5 содер-

жит массу возможностей, для работы которых страница должна предоставляться веб-сервером. (Если попытаться открыть страницу с такими возможностями с локального жесткого диска, они могут отработать с непредвиденным результатом или же не работать вообще.) Просматривая пример с указанного веб-сайта, вы сможете увидеть, как он должен работать, перед тем, как загрузить страницу себе на компьютер и начать экспериментировать с ней.

Примечание

Не волнуйтесь о примерах, которые нужно запускать с веб-сервера — в книге предупреждается о таких примерах.

I

ЧАСТЬ I

Знакомство с новым языком

1

ГЛАВА 1

Представляем HTML5

Историю развития языка HTML можно сравнить с детективным рассказом в том, что в ней есть свой неожиданный поворот событий, в результате которых появилась новая версия языка — HTML5.

Предполагалось, что язык HTML уйдет в небытие, не дожив до XXI столетия. Организация W3C (World Wide Web Consortium, Консорциум Всемирной паутины), которая занимается разработкой и внедрением официальных стандартов Всемирной паутины, забросила язык HTML в далеком 1998 г., считая его не способным на дальнейшее выживание. Свои надежды на будущее консорциум W3C возлагал на модернизированного наследника HTML — язык XHTML. Но язык HTML не умер. Его "подобрала" группа программистов-аутсайдеров и не только возвратила его к жизни, но и заложила основу для новых возможностей, которые мы с вами и исследуем в этой книге.

В данной главе мы выясним, почему язык HTML был брошен умирать от старости и как он был возвращен к жизни. Мы узнаем основные принципы и возможности языка HTML5, а также рассмотрим тернистую проблему поддержки этого языка разными браузерами. Кроме этого, мы также рассмотрим настоящий документ HTML5, как его самую простую форму, так и более практический шаблон, который можно использовать в качестве отправной точки для создания любого веб-сайта.

История языка HTML5

Как вы знаете, HTML — это язык для создания веб-страниц. Ключевая идея языка HTML — организация содержимого с помощью *элементов* — не претерпела никаких изменений с самых ранних времен Всемирной паутины. Более того, даже очень старые веб-страницы без проблем обрабатываются в наиболее современных браузерах (включая и те, которые не существовали на момент создания этих страниц, например Firefox или Chrome).

Но успех и почтенный возраст также несут с собой определенные существенные угрозы. Что и случилось с языком HTML — в 1988 г. консорциум W3C прекратил

его поддержку и попытался заменить его языком на основе языка XML — XHTML 1.0.

Язык XHTML 1.0: строго по правилам

В стандарте XHTML используются те же синтаксические соглашения, что и в HTML, но в нем ужесточены требования к следованию установленным правилам. Большая часть отступлений от правил разметки, которые сходят с рук в традиционном HTML, попросту неприемлемы в XHTML.

Например, допустим, что вы хотите выделить курсивом последнее слово заголовка следующим образом:

```
<h1>Из жизни <i>уток</i></h1>
```

Но при этом вы случайно поменяли местами два последних тега:

```
<h1>Из жизни <i>уток</h1></i>
```

Когда браузер сталкивается с этой слегка подпорченной разметкой, он в состоянии "понять", что вы действительно имели в виду, и без малейших претензий выделяет последнее слово курсивом. Но несовпадающие теги нарушают официальные правила XHTML. Если проверить эту страницу в валидаторе формата XHTML (или открыть ее в какой-либо программе для разработки веб-сайтов, например Dreamweaver), то будет выведено сообщение, указывающее на эту ошибку. В отношении разработки веб-сайтов это полезно, т. к. позволяет отловить небольшие ошибки, вследствие которых страница может отображаться по-разному в различных браузерах, или которые могут вызвать более серьезные проблемы при попытке отредактировать страницу с целью ее улучшения.

Поначалу XHTML пользовался большим успехом. Профессиональные веб-разработчики, раздосадованные индивидуальными особенностями отображения страниц браузерами и вседозволенностью в области веб-разработки, скопом переходили на XHTML. При этом они были вынуждены совершенствовать свои навыки работы и отказываться от использования значительного количества полусырых возможностей форматирования, предоставляемых HTML. Но многие из предполагаемых преимуществ XHTML — такие как функциональная совместимость с инструментами XML, облегчение обработки страниц автоматизированными программами, переносимость на мобильные платформы, а также расширяемость самого языка XHTML — так никогда и не материализовались.

Тем не менее XHTML стал стандартом для большинства серьезных веб-разработчиков. Но в то время, как все казались достаточно довольными этим языком разметки, у него был один скелет в шкафу: хотя браузеры понимали разметку XHTML, они не обеспечивали строгую проверку отступлений от правил, требуемую этим стандартом. Это означает, что страница может быть создана с нарушением правил XHTML, но браузеры и глазом не моргнут при ее обработке. Более того, ничто не могло помешать веб-разработчику бросить в одну кучу небрежно написанную разметку и устаревшее HTML-содержимое и назвать все это страницей XHTML. Ни один браузер на планете не имел бы никаких претензий при обработке

такой страницы. И это обстоятельство вызывало у людей, ответственных за стандарт XHTML, глубокое чувство тревоги.

XHTML 2: неожиданный провал

Эта проблема должна была, по идее, быть решенной в следующей версии — XHTML 2. В ней нужно было ужесточить правила обработки ошибок, которые бы заставляли браузеры не обрабатывать страницы, не отвечающие стандарту XHTML 2. В XHTML 2 также избавились от многих странностей и соглашений, унаследованных от HTML. Например, система нумерации заголовков (`<h1>`, `<h2>`, `<h3>` и т. д.) была заменена одним элементом `<h>` с уровнем обозначаемого им заголовка, зависящего от местонахождения этого элемента в веб-странице. Подобным образом элемент `<a>` был заменен возможностью, позволяющей веб-разработчикам преобразовывать любой элемент в ссылку, а вместо атрибута `alt` элемента `<img>` был предложен новый способ предоставления альтернативного содержимого.

Изменения такого рода были типичными для XHTML 2. С теоретической точки зрения, они делали язык более аккуратным и были более понятными. Но с практической стороны, они вынуждали всех изменить свой подход к созданию веб-страниц (не говоря об обновлении уже созданных веб-страниц), не добавляя при этом никакой новой функциональности, чтобы оправдать всю эту работу. Попутно из XHTML 2 было удалено несколько удобных элементов, которые все еще пользовались популярностью среди веб-разработчиков, таких как `<b>` (полужирный текст), `<i>` (текст курсивом) и `<iframe>` (для вложения одной веб-страницы в другую).

Но, возможно, худшей проблемой оказалась чрезвычайно медленная скорость внесения изменений. Разработка XHTML 2 тащилась в течение пяти лет, и интерес разработчиков к этому стандарту медленно угасал.

HTML5: возвращение к жизни

Приблизительно в то же самое время (начиная с 2004 г.) группа разработчиков начала рассматривать будущее Всемирной паутины в другом ракурсе. Вместо того чтобы попытаться разобраться, что было неправильным (или просто "грязным" с философической точки зрения) в HTML, они сфокусировались на том, чего в нем не хватало, что хотели бы иметь веб-разработчики для воплощения своих идей.

В конце концов, HTML зародился как инструмент для отображения документов. С добавлением языка сценариев JavaScript HTML преобразовался в систему для разработки веб-приложений, таких как поисковые движки, онлайновые магазины, картографические инструменты, средства чтения электронной почты и многие другие. Но в то время как искусное веб-приложение может делать много чего впечатляющего, создать такое приложение — задача не из легких. Большинство разработчиков использует для этого мешанину кода JavaScript, один или несколько популярных инструментариев JavaScript, а также веб-приложение, исполняемое на веб-сервере. Добиться правильного и единообразного взаимодействия всех этих

составляющих на разных браузерах — сложная задача. Даже когда все наконец работает, нужно постоянно "присматривать за скотчем и скобами", которые удерживают всю конструкцию.

Такая ситуация вызывала особенную озабоченность среди разработчиков браузеров, поэтому группа дальновидных разработчиков из компании Opera Software (создатели браузера Opera) и компании Mozilla Foundation (создатели браузера Firefox) начали агитировать за включение в XHTML больше возможностей, ориентированных на разработчиков. Когда их попытки не увенчались успехом, компании Opera, Mozilla и Apple создали группу WHATWG (Web Hypertext Application Technology Working Group, рабочая группа по технологии гипертекстовых веб-приложений) с целью работы над новыми решениями.

Группа не ставила перед собой задачу заменить HTML, ее целью было плавное расширение языка, и причем такое, чтобы расширения были обратно совместимыми. Надо сказать, что самая ранняя версия работы этой группы включала две спецификации расширений — Web Applications 1.0 и Web Forms 2.0. В конечном итоге эти стандарты эволюционировали в HTML5.

Примечание

Предполагается, что число 5 в названии HTML5 означает: данный стандарт является продолжением стандарта HTML (последней версией стандарта HTML перед XHTML была версия 4.01). Это, конечно же, не совсем верно, т. к. HTML5 поддерживает все разработки, существовавшие в области создания веб-страниц в течение десяти лет после выпуска HTML 4.01, включая строгий синтаксис в стиле XHTML (если разработчики желают использовать его), а также множество инноваций для JavaScript. Тем не менее это название делает ясным следующее: язык HTML5 *может* поддерживать соглашения XHTML, но *требует* следования правилам HTML.

К 2000 г. все интересное происходило в лагере группы WHATWG. После некоторого периода болезненных размышлений организация W3C решила распустить работающую над XHTML 2 группу и работать вместо этого над формализацией стандарта HTML5. На этом этапе первоначальный стандарт HTML5 был разделен на более управляемые части, и многие из его функциональных возможностей стали отдельными стандартами *(см. врезку. "На профессиональном уровне. Что входит в состав HTML5?" далее)*.

Совет

Ознакомиться с официальной версией стандарта HTML5 организации W3C можно на веб-сайте по адресу: **www.w3.org/TR/html5**.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Что входит в состав HTML5?

Браузеров, "поддерживающих" HTML5, не существует. Вместо этого, каждый браузер поддерживает постепенно расширяющееся подмножество возможностей HTML5. Этот подход является как хорошим, так и плохим. Хорош он потому, что браузеры могут быстро реализовать готовые части стандарта HTML5, пока другие возможности продолжают развиваться. А плох он тем, что заставляет веб-разработчиков беспокоиться о проверке, поддерживает ли конкретный браузер каждую функциональную возможность, которую они хотят использовать. (Трудные и не очень трудные методы такой проверки рассматриваются далее в этой книге.)

Далее приводится список и короткое описание основных категорий функциональных возможностей, охватываемых HTML5.

- **Ядро HTML5.** Эта часть HTML5 составляет официальную версию спецификации организации W3C. Она содержит новые семантические элементы *(см. главы 2 и 3)*, новые и усовершенствованные элементы управления для веб-форм *(см. главу 4)*, поддержку аудио и видео *(см. главу 5)*, а также холст для рисования с помощью JavaScript *(см. главы 6 и 7)*. В эту категорию входит большинство функциональных возможностей, которые наилучшим образом поддерживаются браузерами.
- **Ранние возможности HTML5.** Это возможности, которые были реализованы в первоначальной спецификации HTML5, подготовленной группой WHATWG. Большинство из них — это спецификации для возможностей, требующих JavaScript и поддерживающих развитые веб-приложения. Наиболее важными являются локальное хранение данных *(см. главу 9)*, приложения, работающие в автономном режиме *(см. главу 10)*, и обмен сообщениями *(см. главу 11)*, но также несколько других, которые рассматриваются в этой книге.
- **Возможности, иногда называемые HTML5.** Это возможности следующего поколения, которые часто считаются частью HTML5, хотя они никогда не входили в стандарт HTML5. Эта категория включает спецификацию CSS3 *(см. главу 8)* и геолокацию *(см. главу 12)*.

Путаница с этими стандартами создается не только ничего не сведущими менеджерами и авторами статей о технологиях. Даже сама организация W3C размывает границы между "настоящим" HTML5 (как определено этим стандартом) и "маркетинговой" версией (которая включает все новшества и кухонную раковину). Например, официальный веб-сайт организации W3C по логотипам (**www.w3.org/html/logo**) призывает разработчиков генерировать логотипы HTML5, продвигающие стандарты CSS3 и SVG-2, разработка которых велась задолго до появления HTML5.

HTML: живой язык

В результате перехода поддержки HTML сначала от организации W3C к группе WHATWG, а потом обратно, возникла довольно необычная ситуация. Технически организация W3C отвечает за определение, что является официальным HTML5, а что — нет. Но в то же самое время группа WHATWG продолжает свою работу, придумывая будущие возможности HTML. Только теперь она называет его не HTML5, а просто HTML, объясняя это тем, что HTML будет продолжать существовать, как *живой язык*.

Так как HTML является живым языком, то HTML-страница никогда не устареет и не перестанет работать. Для HTML-страниц никогда не потребуется номер версии (даже в блоке указания типа документа `<doctype>`), а веб-разработчикам никогда не понадобится обновлять свою разметку от одной версии языка к другой, чтобы она работала на новых браузерах.

Так как HTML — это живой язык, новые возможности (и новые элементы) можно добавлять к стандарту HTML в любое время. Некоторые разработчики могут решить использовать эти возможности в своих веб-страницах, а некоторые разработчики веб-браузеров — поддерживать их в своих продуктах. Но возможности никогда не будут привязаны к какой-либо конкретной версии стандарта.

Обычно, когда веб-разработчики слышат о новшествах, первым делом они приходят в полный ужас. В конце концов, кому охота иметь дело с поддержкой стандарта, требования которого варьируются в широком диапазоне, где разработчикам нужно выбирать возможности на основе вероятности поддержки этих возможностей? Но немного поразмыслив, большинство веб-разработчиков нехотя соглашается: хорошо это или плохо, но именно таким образом браузеры работают сегодня и работали с самого начала существования Всемирной паутины.

Как упоминалось ранее, сегодняшние браузеры не имеют никаких претензий к любой мешанине поддерживаемых возможностей. Например, можно создать XHTML-страницу по самому последнему слову этой технологии, а потом добавить в нее такой ужасно устаревший элемент, как `<marquee>` (используемый для создания "бегущей строки"), и ни один из браузеров не будет жаловаться на такую страницу. Также существуют и дыры в поддержке браузерами даже самых старых стандартов. Например, разработчики браузеров начали внедрять спецификацию CSS3, прежде чем была прекращена поддержка спецификации CSS2, и многие возможности CSS2 были потом заброшены. И есть немалая ирония в том, что как только HTML вступает в новую, инновационную эпоху, он возвращается на круги своя — к своим истокам.

Совет

С текущим, продолжающимся развиваться, черновым вариантом стандарта HTML, включающим материал, который называется HTML5, а также небольшой, но постоянно развивающийся набор новых, неподдерживаемых возможностей, можно ознакомиться на сайте **http://whatwg.org/html**. А последние, менее формальные новости по HTML можно узнать в блоге группы WHATWG по адресу **http://blog.whatwg.org**.

Три основных принципа HTML5

Итак, вам, наверное, уже не терпится начать работать над настоящей страницей HTML5. Но прежде чем приступить к этому, сначала полезно заглянуть в мысли создателей HTML5. Понимая философию, на которой зиждется этот язык, вам будет намного легче вникать в странности, сложности и разбираться со случайными трудностями, с которыми вы столкнетесь в этой книге.

Принцип 1. Не рвити Паутину

Требование "Не рвать Паутину" означает, что стандарт не должен вносить изменения, которые сделают нерабочими веб-страницы других разработчиков. Но такое случается редко.

"Не рвать Паутину" *также* означает, что стандарт не должен мимоходом изменять правила и считать устаревшими совершенно нормальные современные веб-страницы (если они продолжают работать). Например, XHTML 2 порвал Паутину, т. к. требовал немедленно коренным образом изменить подход к написанию веб-страниц. Да, благодаря встроенной в браузеры поддержке обратной совместимости старые страницы продолжали бы работать. Но чтобы подготовиться к будущему и

поддерживать свою веб-страницу, разработчикам потребовалось бы затратить множество часов, исправляя запрещенные в XHTML 2 "ошибки".

В HTML5 совсем другая философия. Все, что было правильным до HTML5, остается правильным и в HTML5. Таким образом, все, что было правильным в HTML 4.01, остается правильным в HTML5.

Примечание

В отличие от предыдущих стандартов, в HTML5 не просто указывается разработчикам браузеров, что им поддерживать, но также документируется и формализуется способ их работы, которого они *уже придерживаются*. Так как в стандарте HTML5 документируется действительность, а не просто предписывается куча идеальных правил, он может стать наиболее поддерживаемым веб-стандартом, который когда-либо существовал.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Как HTML5 обрабатывает устаревшие элементы

Поддержка в HTML5 всех соглашений HTML означает, что он поддерживает многие возможности, которые в настоящее время считаются устаревшими. В число этих возможностей входят такие элементы форматирования, как `<font>`, ненавидимые многими элементы для создания специальных эффектов, такие как `<blink>` и `<marquee>`, а также громоздкая система HTML-фреймов.

Такая непредвзятость вводит в замешательство многих начинающих изучать HTML5. С одной стороны, в HTML5 с полным правом должны быть запрещены все эти устаревшие элементы, которые в течение многих лет не упоминались в официальных спецификациях. Но с другой стороны, современные браузеры без лишнего шума продолжают поддерживать эти элементы, а HTML5 должен отражать, как в действительности работают веб-браузеры. Что же стандарту остается делать?

Эта проблема решается путем разделения спецификации HTML5 на две отдельные части. Первая часть, которая рассматривается в этой книге, предназначена для веб-разработчиков. Им нужно избегать неряшливых привычек и устаревших элементов. Проверить следование требованиям этой части стандарта HTML5 можно с помощью валидатора HTML5.

А вторая часть (намного объемнее первой) предназначена для разработчиков веб-браузеров. Им нужно поддерживать все, что когда-либо существовало в HTML, с тем, чтобы обеспечить своим браузерам обратную совместимость.

В идеальном случае, стандарт HTML5 должен содержать достаточно информации, позволяющей кому угодно создать браузер с чистого листа и обеспечить его полную совместимость с современными браузерами независимо от типа обрабатываемой им разметки — старой или новой. В этой части стандарта указывается, как браузеры должны обрабатывать устаревшие элементы, употребление которых официально не советуется, но которые продолжают поддерживаться.

Кстати, спецификация HTML5 также формализует способ обработки браузерами разнообразных ошибок (например, отсутствующие или несовпадающие теги). Это важный аспект, т. к. таким образом обеспечивается единообразная обработка дефектной страницы разными браузерами, даже в случае таких тонких проблем, как способ создания модели страницы в системе DOM[1] (дерево объектов в памяти, представляющее страницу, доступную коду JavaScript). Для создания этой объемной, трудоемкой части

[1] Document Object Model — объектная модель документа.

стандарта создатели HTML5 провели всеохватывающее тестирование на современных браузерах, чтобы выяснить их незадокументированное поведение при обработке ошибок. Наконец, они изложили все это на бумаге.

Принцип 2. Асфальтируйте тропинки

Тропинка представляет собой неровный, протоптанный путь, позволяющий людям добраться из одной точки в другую. Тропинки существуют, т. к. они используются. Тропинка может быть не лучшим путем, но на определенном этапе она является самым практическим работающим решением.

Стандарт HTML5 задается целью стандартизировать эти неофициальные, но широко применяемые, методы. Возможно, результат этого подхода не будет таким аккуратным, как прокладка новой заасфальтированной дороги с применением последних технологий, но у него больше шансов на успех. Причина кроется в том обстоятельстве, что переход к использованию новых методов может оказаться не под силу или не представлять интереса для веб-разработчика среднего уровня. Что еще хуже, новые методы могут не работать в старых браузерах, которыми пользуются посетители веб-страницы. Стандарт XHTML 2 пытался заставить людей не пользоваться тропинками, но потерпел в этом предприятии сокрушающее поражение.

Примечание

Асфальтирование тропинок несет очевидную выгоду: в нем используются упрочившиеся методы, уже поддерживающиеся браузерами на некотором уровне. Веб-разработчики в любое время отдадут предпочтение неряшливому практическому решению, которое всегда работает на всех браузерах, обеспечивая тем самим более обширную аудиторию, чем искусно разработанной новой возможности, работающей лишь на 70% браузеров.

Но подход с асфальтированием тропинок также требует определенных компромиссов. Иногда это означает признание широко поддерживаемой, но плохо разработанной возможности. Одним из примеров этому может служить операция drag and drop (перетащить и бросить) *(см. разд. "Чтение файла изображения" главы 9)*, которая полностью основана на поведении, созданном компанией Microsoft для своего браузера IE 5. Хотя в настоящее время возможность "перетащить и бросить" поддерживается во всех браузерах, ее ненавидят все разработчики из-за ее громоздкости и большой сложности. Поддержка этой возможности в HTML5 вызвала у некоторых веб-разработчиков жалобы на то, что "HTML5 не только поддерживает плохие свойства, но и устанавливает стандарт для них".

Принцип 3. Будьте практичными

Это простой принцип: все изменения должны служить практической цели. И чем более трудоемкое изменение, тем большей должна быть ожидаемая от нее отдача. Веб-разработчикам могут быть больше по душе тщательно разработанные, единообразные стандарты без странностей, но это недостаточно веская причина, чтобы менять язык, на котором уже создано несколько миллиардов документов. Конечно

же, кто-то должен еще решить, чьи интересы являются более важными. Для решения этого вопроса хорошо бы посмотреть, что веб-страницы уже делают или пытаются делать.

Например, третьим самым популярным веб-сайтом в мире (на момент написания этой книги) был сайт YouTube. Но так как до HTML5 в HTML не было настоящих возможностей поддержки видео, то в своей работе создателям этого сайта пришлось использовать подключаемый к браузеру Flash-модуль. Это решение работает на удивление хорошо, т. к. такой Flash-модуль установлен, как правило, на всех подключенных к Интернету компьютерах. Но иногда случаются исключения из этого правила: на корпоративных компьютерах может быть запрещена установка Flash-модуля, а устройства Apple (такие как iPhone или iPad) вообще не поддерживают его. И несмотря на то, на скольких компьютерах может быть установлен Flash-модуль, есть хорошее основание для расширения стандарта HTML, чтобы он поддерживал напрямую одно из самых распространенных применений веб-страниц — просмотр видео.

Подобным образом мотивируется стремление добавить к HTML5 дополнительную поддержку метода "перетащить и бросить" для интерактивных элементов, редактируемого HTML-содержимого, рисования на двумерном холсте и т. п. Не нужно далеко ходить, чтобы найти веб-страницы, в которых все эти возможности применяются уже сейчас. В одних это делается с помощью подключаемых модулей типа Adobe Flash или Microsoft Silverlight, а в других — с помощью библиотек JavaScript или (что более трудоемко) посредством специально созданных страниц JavaScript. Почему же тогда не добавить в стандарт HTML официальную поддержку этих возможностей и не обеспечить их единообразную работу на всех браузерах?

Примечание

Подключаемые модули типа Flash не исчезнут в одночасье (или даже в течение нескольких лет). Несмотря на многие инновации, создание сложных графических приложений в HTML5 требует намного больше работы, чем использование для этих целей подключаемых модулей (см., например, игры для браузеров на сайте **www.flasharcade.com**). Но конечная цель стандарта HTML5 ясна: позволить веб-сайтам предоставлять видео, развитую интерактивность, а также кучу разных других примочек, не прибегая к помощи подключаемых модулей.

Первое знакомство с разметкой HTML5

Рассмотрим один из простейших документов HTML5. Он начинается с указания типа документа с помощью специального кода описания типа документа (значение этого кода объясняется в *следующем разделе*), после чего дается название документа, а потом идет его содержимое. В данном случае содержимое состоит из одного абзаца текста.

```
<!DOCTYPE html>
<title>A Tiny HTML Document</title>
<p>Let's rock the browser, HTML5 style.</p>
```

Или, что то же самое[1]:

```
<!DOCTYPE html>
<title>Крошечный документ HTML</title>
<p>Дадим встряску браузеру в стиле HTML5.</p>
```

Результат обработки этого документа браузером показан на рис. 1.1.

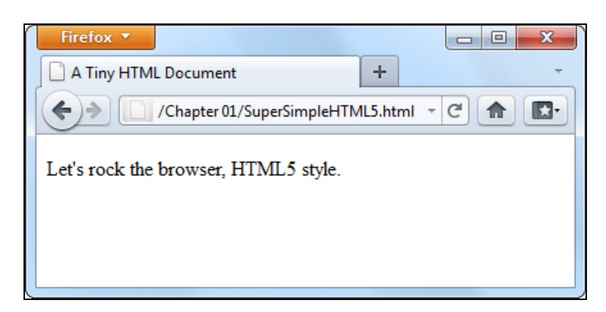


Рис. 1.1. Отображение простейшего HTML5 документа в браузере

Этот насколько простой документ можно упростить еще больше. Например, конечный тег </p> вообще-то не является обязательным в стандарте HTML5, т. к. браузеры знают, как закрывать все открытые элементы в конце документа (а стандарт HTML5 узаконивает это поведение). Но подобное срезание углов вместо упрощения делает разметку более сложной для понимания и может вызвать неожиданные ошибки.

Стандарт HTML5 также разрешает удалить элемент <title>, если информация о названии страницы предоставляется посредством какого-либо другого механизма. Например, при отправлении HTML-документа в сообщении электронной почты его название можно вставить в поле **Тема** сообщения, а остальной код разметки — тип документа и содержимое — поместить в поле для сообщения. Но это, очевидно, особый случай.

Далее нам захочется облечь плотью этот скелетистый HTML5-документ. Большинство веб-разработчиков придерживается мнения, что использование традиционных разделов <head> и <body> полезно для облегчения восприятия документа, т. к. они четко разделяют информацию о странице (заголовок страницы) и само содержимое (основная часть страницы). Такая структура особенно полезна, когда к странице добавляются сценарии, таблицы стилей и метаэлементы, как показано в следующем листинге:

```
<!DOCTYPE  html>
<head>
  <title>Крошечный документ HTML</title>
</head>
<body>
  <p>Дадим встряску браузеру в стиле HTML5.</p>
</body>
```

[1] Иногда фрагменты кода будут представлены на русском языке. — *Ред.*

Как всегда, отступ (в данном случае в начале третьей и шестой линий кода) является абсолютно необязательным. В этом примере отступ используется с целью облегчить понимание структуры страницы с первого взгляда.

Наконец, весь документ (за исключением строки `doctype`) можно облачить в традиционный элемент `<html>`, как показано в следующем листинге:

```
<!DOCTYPE html>
<html>
<head>
  <title>Крошечный документ HTML</title>
</head>
<body>
  <p>Дадим встряску браузеру в стиле HTML5.</p>
</body>
</html>
```

Вплоть до HTML5 в каждой версии официальной спецификации HTML требовалось использование элемента `<html>`, несмотря на то, что наличие этого элемента никаким образом не влияет на обработку браузером остального кода страницы. В HTML5 использование этого элемента оставлено полностью на личное усмотрение разработчика.

Примечание

Использование элементов `<html>`, `<head>` и `<body>` является просто вопросом стиля. Страница без этих элементов будет работать отличнейшим образом даже на старых браузерах, которые и слыхом не слыхивали ни о каком HTML5. Практически, браузер автоматически предполагает наличие этих элементов. Поэтому, если посмотреть на модель DOM (набор программных объектов, представляющих страницу) страницы с помощью сценария JavaScript, она будет содержать объекты для элементов `<html>`, `<head>` и `<body>`, даже если разработчик и не использовал их.

На данном этапе этот пример страницы является чем-то средним между самым простым HTML5-документом и расширенной отправной точкой практической веб-страницы HTML5. В последующих разделах мы добавим к нему остальные необходимые элементы и копнем глубже в разметку.

Описание типа документа HTML5

В первой строке каждого HTML5-документа всегда дается описание *типа документа*. Это описание ясно указывает, что далее следует HTML5-содержимое, и выглядит следующим образом:

```
<!DOCTYPE html>
```

Первое, что бросается в глаза в описании типа документа HTML5, — это его поразительная простота. Сравните его, например, с неуклюжим описанием типа документа, который требуется использовать веб-разработчикам при работе со строгим XHTML 1.0:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Даже профессиональные веб-разработчики были вынуждены вставлять описание типа документа XHTML методом копирования и вставки из другого документа. А описание типа документа HTML5 короткое, четкое и легко вводится вручную.

Описание типа документа HTML5 также примечательно тем, что оно не содержит номера официальной версии HTML (5 для HTML5). В нем просто указывается, что страница является HTML-страницей. Это соответствует новой концепции HTML5, как живого языка. Добавленные в HTML новые возможности автоматически доступны для размещения на странице, не требуя для этого изменений в описании типа документа.

Все это порождает непростой вопрос: если HTML5 — живой язык, то зачем тогда для страницы вообще нужно описание типа документа?

Ответ на этот вопрос таков: описание типа документа продолжает использоваться по историческим причинам. При обработке страницы с отсутствующим описанием типа документа большинство браузеров (включая Internet Explorer и Firefox) переходят в *режим совместимости* (quirks mode). В этом режиме они пытаются отобразить страницу с учетом ошибок в правилах, которые использовались в более ранних версиях. Проблема с этим состоит в том, что режим совместимости одного браузера может отличаться от режима совместимости другого браузера, вследствие чего страницы, разработанные для одного браузера, на другом браузере будут, скорее всего, отображаться с ошибками, такими как неправильный размер шрифта, нарушенная структура оформления и т. п.

А обнаружив на странице описание типа документа, браузер знает, что обработку этой страницы требуется выполнять, следуя более строгим правилам *режима стандартов* (standards mode), который обеспечивает единообразное форматирование и структуру страницы при ее отображении любым современным браузером. За некоторыми исключениями, браузеру совершенно безразлично, *какой именно* тип документа указан в описании. Он просто проверяет, что страница имеет *какое-либо* описание типа документа. Описание типа документа HTML5 просто самое короткое действительное описание типа документа, которое задействует режим стандартов браузера.

Совет

Описание типа документа HTML5 задействует режим стандартов во всех браузерах, имеющих этот режим, включая браузеры, которые ничего не ведают об HTML5. Поэтому вы можете начать использовать описание типа документа HTML5 прямо сейчас во всех создаваемых вами страницах, даже если вы вынуждены отложить использование многих возможностей HTML5 с меньшим уровнем поддержки.

Хотя описание типа документа в первую очередь предназначено для указания веб-браузерам, какой режим обработки страниц использовать, другие агенты могут также проверять его. В число этих агентов входят HTML5-валидаторы, поисковые движки, инструменты разработки, а также иные разработчики (когда они пытаются вычислить тип разметки, используемый данной страницей).

Кодировка символов

Кодировка — это стандарт, указывающий компьютеру, каким образом преобразовывать текст в последовательность байтов при записи текста в файл (а также, как выполнять обратное преобразование при открытии файла). По историческим причинам в мире существует множество разных кодировок. В настоящее время практически на всех веб-сайтах используется компактная и быстрая кодировка UTF-8, поддерживающая все символы других алфавитов, которые вам когда-либо могут потребоваться.

Веб-серверы часто настраивают, чтобы сообщать посещающим их браузерам, что предлагаемые сервером страницы имеют определенную кодировку. Но нельзя быть уверенным, что веб-сервер, на котором вы планируете разместить свой веб-сайт, будет настроен таким образом (если только это не ваш собственный сервер). Кроме этого, попытка браузера в таком случае предположить наиболее вероятную используемую кодировку может претерпеть неудачу по причине какого-либо малоизвестного вопроса безопасности. По этим причинам всегда следует вставлять информацию об используемой кодировке в разметку страницы.

HTML5 делает эту задачу легкой. Для этого нужно лишь вставить элемент `<meta>` в самом начале блока `<head>` (или, если элемент `<head>` не используется, сразу же после кода описания типа документа). Вот пример использования элемента `<meta>`:

```
<head>
  <meta charset="utf-8">
  <title>Крошечный документ HTML</title>
</head>
```

Инструменты для веб-разработки, такие как Dreamweaver или Expression Web, вставляют этот элемент автоматически при создании страницы. Эти инструменты также обеспечивают сохранение файлов в кодировке UTF. Но при создании веб-страницы с помощью обычного текстового редактора, чтобы обеспечить сохранение файлов в правильной кодировке, может потребоваться выполнить дополнительные шаги. Например, HTML-файл, редактируемый с помощью программы Блокнот (в Windows), нужно сохранять посредством команды **Сохранить как** и при этом выбрать кодировку UTF-8 в раскрывающемся списке **Кодировка** внизу диалогового окна. А в текстовом редакторе TextEdit (в Mac OS) сначала нужно выбрать **Формат | Обычный текст**, а потом в раскрывающемся списке **Кодировка обычного текста** диалогового окна **Сохранить как** выбрать опцию **Unicode (UTF-8)**.

Язык

Считается хорошим тоном указывать *естественный язык* веб-страницы. Эта информация может быть иногда полезной для других, например, поисковые движки могут использовать ее для фильтрации результатов поиска, чтобы возвратить только страницы на указанном языке.

Чтобы указать язык для какого-либо содержимого, используется атрибут `lang` в любом элементе разметки с заданием кода требуемого языка. Код для русского

языка — ru, а для английского — en. Коды для других языков можно узнать на сайте **http://people.w3.org/rishida/utils/subtags**.

Вставить в веб-страницу информацию о языке легче всего через элемент <html>:

```
<html lang="en">
```

Информация о языке также может быть полезной, если страница содержит текст на разных языках, который нужно прочитать с помощью программы чтения экранного текста. В таком случае атрибут lang с кодом соответствующего языка вставляется в нужном месте документа, например, в элементы <div>, охватывающие текст на разных языках. Таким образом, программа чтения экранного текста способна определить, какой текст она может читать.

Добавление таблицы стилей

Практически в каждой странице должным образом разработанного профессионального веб-сайта используются таблицы стилей. Для указания требуемой таблицы стилей используется элемент <link> в блоке <head> документа HTML5:

```
<head>
   <meta charset="utf-8">
   <title>Крошечный документ HTML</title>
   <link href="styles.css" rel="stylesheet">
</head>
```

Этот способ похож на указание таблиц стилей в традиционных HTML-документах, но немного проще. Так как существует единственный язык каскадных таблиц стилей — CSS, то в добавление атрибута type="text/css", который требовался ранее, больше нет надобности.

Добавление JavaScript-кода

Язык сценариев JavaScript изначально создавался как средство для трудоемкого способа придания блеска и обаяния скучным веб-страницам. В настоящее время основная область применения JavaScript сместилась с разработки прибамбасов интерфейса на разработку нестандартных веб-приложений, включая сверхэффективных клиентов электронной почты, текстовых редакторов и картографических движков, которые исполняются непосредственно в браузере.

Код JavaScript вставляется в документ HTML5 по большому счету таким же способом, как и в традиционную HTML-страницу. В следующем листинге приводится пример вставки в веб-документ кода JavaScript по ссылке на внешний файл:

```
<head>
  <meta charset="utf-8">
  <title>Крошечный документ HTML</title>
  <script src="scripts.js"></script>
</head>
```

Атрибут `language="JavaScript"` не является обязательным, т. к. если не указан какой-либо другой язык сценариев (а поскольку JavaScript — единственный широко-поддерживаемый язык сценариев для HTML, то вероятность такого развития ничтожно мала), браузеры автоматически предполагают, что используется JavaScript. Но даже ссылаясь на внешний файл с кодом JavaScript, все равно *нужно* помнить о закрывающем теге `</script>`. Если упустить этот тег по недосмотру или при попытке укоротить код, используя синтаксис пустых элементов, то страница не будет работать должным образом.

Если вы уделяете много времени тестированию своих страниц с JavaScript в Internet Explorer, может быть полезным добавление метки MOTW (Mark of the Web, метка особенности сети) в блок `<head>` сразу же после строки кодировки. Делается это таким образом:

```
<head>
  <meta charset="utf-8">
  <!-- saved from url=(0014)about:internet -->
  <title>Крошечный документ HTML</title>
  <script  src="scripts.js"></script>
</head>
```

Эта строка кода указывает Internet Explorer обрабатывать страницу таким образом, как будто бы она была загружена с удаленного веб-сайта. В противном случае IE переключается в особый режим блокировки, выводит предупреждение безопасности в строке сообщений и отказывается исполнять любой код JavaScript до тех пор, пока вы не нажмете кнопку **Разрешить заблокированное содержимое**.

Все другие браузеры не обращают внимания на метку MOTW и используют одни и те же настройки безопасности как для страниц, загружаемых с удаленных веб-сайтов, так и для локальных файлов HTML.

Конечный результат

Если вы выполнили все вышеизложенные шаги, созданный вами документ HTML5 должен выглядеть подобным образом:

```
<!DOCTYPE  html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Крошечный документ HTML</title>
  <link href="styles.css"  rel="stylesheet">
  <script  src="scripts.js"></script>
</head>

<body>
  <p>Дадим встряску браузеру в стиле HTML5.</p>
</body>
</html>
```

Пусть это больше не самый короткий документ HTML5, зато неплохая отправная точка для создания любой веб-страницы. И хотя этот пример может показаться тошнотворно скучным, не переживайте по этому поводу, т. к. в следующей главе мы повысим уровень наших разметок до практических страниц, полных тщательно оформленного содержимого, отформатированного с помощью таблиц стилей.

Примечание

Все элементы синтаксиса HTML5, рассмотренные в этом разделе — новое описание типа документа, метаэлемент кодировки, атрибут языка, таблицы стилей и ссылки на код JavaScript — работают как в новых, так и в старых браузерах. Это возможно благодаря тому, что они полагаются на настройки по умолчанию и встроенные схемы исправления ошибок, которые используются во всех браузерах.

Углубленное знакомство с синтаксисом HTML5

Как вы уже узнали, в HTML5 некоторые правила были ослаблены. Это было сделано потому, что создатели HTML5 хотели, чтобы этот язык реальнее отражал действительную работу веб-браузеров. Иными словами, они хотели сократить разрыв между "работающими веб-страницами" и "веб-страницами, правильными с точки зрения стандарта". В следующем разделе мы рассмотрим изменения в правилах более подробно.

Примечание

Конечно же, все еще существуют устаревшие методы, поддерживаемые браузерами, употребление которых абсолютно не одобряется стандартом HTML5. Эти методы можно обнаружить с помощью валидатора HTML5 *(см. разд. "Проверка кода HTML5" далее в этой главе)*.

Ослабленные правила

При нашем первом знакомстве с разметкой HTML5 мы узнали, что использование элементов `<html>`, `<head>` и `<body>` не является обязательным для этой разметки. Но ослабление правил в HTML5 на этом не заканчивается.

В нем также разрешается использовать в тегах как прописные, так и строчные буквы, как в следующем примере:

```
<P>В тегах <EM>можно использовать</eM> как прописные, так и строчные
буквы.</p>.
```

Также можно не использовать закрывающую обратную косую черту в пустых элементах, т. е. элементах без содержимого, таких как `<img>` (изображение), `<br>` (разрыв строки) или `<hr>` (горизонтальная линия). Далее приведены три равнозначных способа вставить разрыв строки:

```
Я не могу<br />
двинуться ни вперед,<br>
ни назад.<br/>
Я застрял.
```

В HTML5 также подверглись изменениям правила для атрибутов. Значения атрибутов больше не требуется брать в кавычки, но только при условии, что они не содержат запретных символов (обычно это символы >, = или пробел). Вот пример использования элемента `<img>` таким образом:

```
<img alt="Туманность Конская голова" src=Horsehead01.jpg>
```

Также разрешены атрибуты без значений. Таким образом, если для установки флажка в XHTML требуется несколько повторяющийся синтаксис:

```
<input type="checkbox" checked="checked" />
```

то в HTML5 это можно делать в традициях HTML 4.01, указывая только одно название атрибута:

```
<input type="checkbox" checked>
```

Но некоторых особенно беспокоит не то, что все это разрешено в HTML5, а то, что не особенно последовательные разработчики могут небрежно использовать как строгие, так и ослабленные правила, иногда даже в одном и том же документе. Но в действительности подобная небрежная разметка возможна и в XHTML. В обоих стандартах обязанность за хороший стиль разметки лежит на веб-разработчике, т. к. браузер скушает все, что ему будет подано.

Далее дается краткое изложение основных принципов хорошего стиля создания разметки HTML5. Здесь же указываются соглашения, применяемые в примерах этой книги, даже если следование этим соглашениям не является обязательным.

- **Использование элементов `<html>`, `<body>` и `<head>`.** В элементе `<html>` удобно размещать определение естественного языка страницы *(см. разд. "Язык" ранее в этой главе)*, а элементы `<head>` и `<body>` позволяют отделить информацию о странице от собственно содержимого страницы.
- Строчные буквы в тегах. Использование строчных букв в тегах не является обязательным, но такие теги намного больше распространены, их легче вводить (т. к. не требуется задействовать клавишу <Shift>), а также не так режут глаз, как теги с прописными буквами.
- Взятие в кавычки значений атрибутов. Значения атрибутов берутся в кавычки потому, что на это есть причина — помочь избежать ошибок, которые в противном случае очень легко допустить. Без кавычек один неправильный символ значения атрибута может испортить всю страницу.

С другой стороны, существуют некоторые старые правила, которые в этой книге игнорируются (что также разрешается делать и вам). В примерах в этой книге пустые элементы не закрываются, т. к. при переходе на HTML5 большинство разработчиков не утруждает себя добавлением лишней косой черты (/). Также нет причины использовать длинную форму атрибутов при одинаковом названии и значении атрибута.

Проверка кода HTML5

Новый расслабляющий подход к правилам в HTML5 может быть вполне по душе одним веб-разработчикам. Других же сама мысль о том, что за фасадом работаю-

щего без сучка, без задоринки браузера может скрываться непоследовательная, полная ошибок разметка, способна лишить сна. Если вы принадлежите ко второму типу, то знайте, что инструмент для проверки правильности разметки, называющийся *валидатором*, может обнаружить код, который не соответствует рекомендуемым стандартам HTML5, даже если браузер и глазом не моргнет при обработке этого кода.

Некоторые из возможных проблем, которые валидатор в состоянии уловить, включают следующие:

- отсутствие обязательных элементов (например, элемента `<title>`);
- отсутствие закрывающего тега;
- неправильно внедренные теги;
- отсутствие атрибутов у тегов, для которых они обязательны (например, атрибута `src` тега `<img>`);
- неправильное расположение элементов или содержимого (например, текста в блоке `<head>`).

Инструменты для разработки веб-страниц, такие как Dreamweaver и Expression Web, оснащены собственными валидаторами, но только самые последние версии поддерживают HTML5. В таком случае можно воспользоваться одним из онлайновых валидаторов. Далее даются инструкции по использованию популярного валидатора от организации W3C:

1. Откройте в своем браузере страницу **http://validator.w3.org** (рис. 1.2).

   Валидатор предложит три способа проверки разметки, каждая на своей вкладке: **Validate by URI** (для страницы, которая уже размещена в Паутине), **Validate by File Upload** (для страницы, сохраненной в файле на вашем компьютере) и **Validate by Direct Input** (для кода, вводимого или вставляемого в окно валидатора).

2. Выберите нужную вкладку и предоставьте свою HTML-разметку.

   - Для метода **Validate by URI** нужно ввести URL проверяемой страницы в поле **Address** (например, **http://www.MySloppySite.com/FlawedPage.html**).
   - Для метода **Validate by File Upload** необходимо загрузить в валидатор требуемый файл с вашего компьютера. Для этого сначала нажмите кнопку **Choose** (Выберите файл) в поле **File** (в браузере Chrome нажмите кнопку **Выбрать файл**). В диалоговом окне **Открыть** выберите требуемый HTML-файл и нажмите кнопку **Открыть**.
   - Метод **Validate by Direct Input** позволяет проверить любой код, помещенный в поле **Enter the Markup to validate**. Для этого метода легче всего будет скопировать код из текстового редактора и поместить его в поле для проверки.

   Прежде чем приступить к проверке кода, можно щелкнуть по ссылке **More Options**, чтобы изменить некоторые параметры, но это нежелательно. В част-

ности, будет лучше предоставить валидатору определить тип документа автоматически. Таким образом валидатор использует описание типа документа, указанное в проверяемой веб-странице. Также лучше предоставить валидатору самому определить кодировку страницы, за исключением страниц с кодировкой, которую валидатор затрудняется определить.

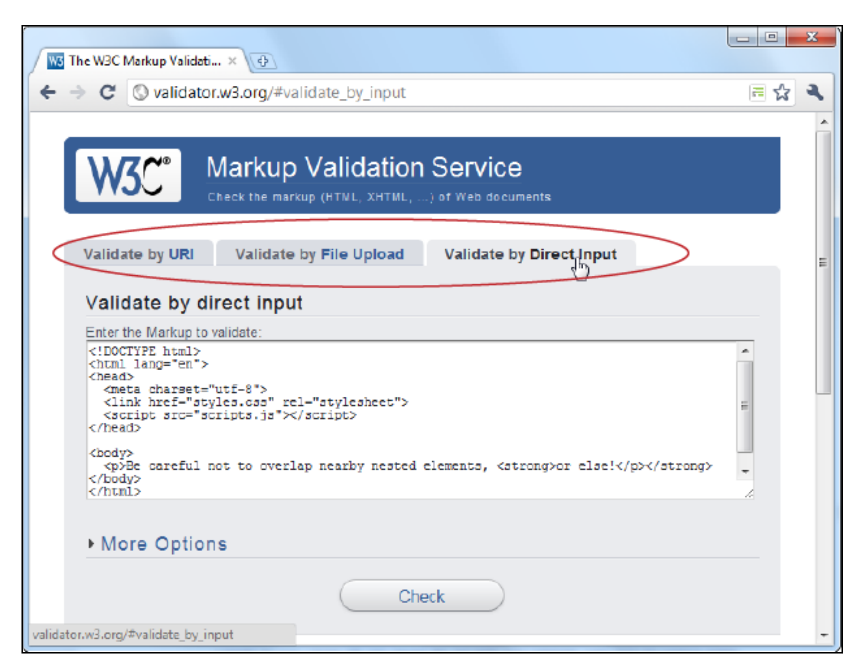


Рис. 1.2. Веб-сайт **http://validator.w3.org** позволяет выполнить проверку разметки HTML тремя разными способами

3. Нажмите кнопку **Check**.

   Ваш код будет отправлен на проверку, и после короткого ожидания в браузере будет выведен отчет с результатами валидации. Если код не прошел проверку, то в отчете будут указаны выявленные валидатором ошибки (рис. 1.3).

Примечание

Даже для полностью правильного HTML-документа в отчете может быть указано несколько предупреждений (хотя полностью безобидных), включая такие, что кодировка была определена автоматически и услуга валидации кода HTML5 является экспериментальной и не совсем доведенной до логического конца.

Как можно видеть на рис. 1.3, валидатор выявил в документе четыре нарушения правил HTML5, являющиеся результатом двух ошибок в коде. Первая ошибка — отсутствует обязательный элемент `<title>`. Вторая — элемент `<p>` закрывается до закрытия вложенного в него элемента `<strong>`. Чтобы исправить эту ошибку, нужно вместо последовательности элементов `</p></strong>` использовать последовательность `</strong> </p>`. Тем не менее, несмотря на эти ошибки, эта разметка правильная, и все браузеры будут отображать эту страницу должным образом.

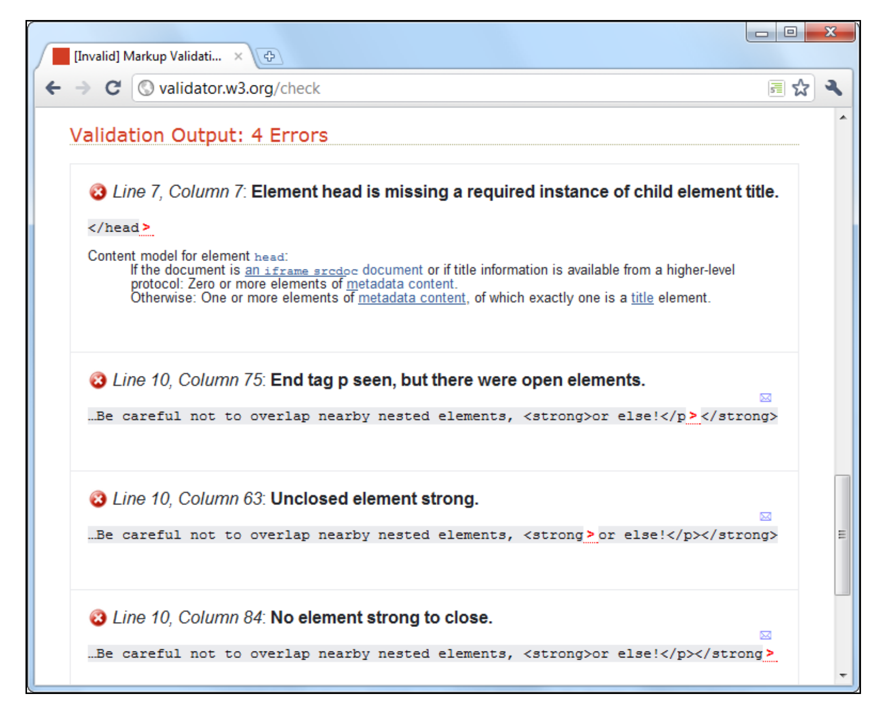


Рис. 1.3. Отчет с результатами проверки

Возвращение XHTML

Как мы уже узнали, восхождение спецификации HTML5 знаменует, по идее, закат предыдущего короля Всемирной паутины — стандарта XHTML. Но действительность не так проста, и поклонникам XHTML не нужно отказываться ни от чего, что им мило в языках разметки предыдущего поколения.

Прежде всего, вспомним, что синтаксис XHTML продолжает существовать. Налагаемые XHTML правила либо продолжают использоваться в качестве руководящих принципов (например, правила правильного вложения элементов), либо поддерживаются в виде необязательных соглашений (например, соглашение об использовании закрывающей косой черты с пустыми элементами).

Но что если вы хотите сделать следование правилам XHTML-синтаксиса обязательным? Возможно, вы беспокоитесь, что вы (или ваши коллеги по работе) неосознанно потихоньку впадете в использование ослабленных соглашений обычного HTML. Чтобы не допустить этого, вам нужно использовать XHTML5; это менее распространенный стандарт, который, по сути, является HTML5, облаченным в ограничения, основанные на XML.

Чтобы сделать документ HTML5 документом XHTML, нужно явно указывать пространство имен XHTML в элементе `<html>`, закрывать каждый элемент, обязательно использовать строчные буквы в тегах и т. д. В следующем листинге приводится пример кода, в котором выполнены все эти требования:

```
<!DOCTYPE  html>
<html  lang="en" xmlns="http://www.w3-org/1999/xhtml">
<head>
  <meta charset="utf-8"/>
  <title>Крошечный документ HTML</title>
  <link href="styles.css" rel="stylesheet"/>
  <script src="scripts.js"></script>
</head>

<body>
  <p>Дадим встряску браузеру в стиле XHTML5.</p>
</body>
</html>
```

Для проверки этого кода требуется валидатор XHTML, который контролирует следование строгим старым правилам XHTML. Валидатор от W3C для этого не подойдет, но зато подойдет валидатор на сайте **http://validator.nu**, где нужно указать требуемый стандарт (т. е. XHTML) в раскрывающемся списке **Preset**. Также нужно установить флажок **Be lax about HTTP Content-Type**, если только вы не вставляете проверяемый код непосредственно в текстовое поле.

Следуя этим шагам, вы сможете создать документ XHTML и выполнить его проверку. Тем не менее *браузеры* все равно будут обрабатывать этот документ, как обычную страницу HTML5, которая просто пыжится походить на XML-документ. Никаких дополнительных правил при обработке такой страницы применять они не будут.

Если же вы хотите, чтобы и браузеры обрабатывали страницу согласно правилам XHTML, то вам нужно настроить свой веб-сервер для подачи страниц с MIME-типом `application/xhtml+xml` или `application/xml`, вместо стандартного типа `text/html`. (Краткую информацию о MIME-типах *см. во врезке "На профессиональном уровне. Основные сведения о MIME-типах" главы 5.)* Но прежде чем звонить администратору вашего веб-хостинга и давать ему эти инструкции, имейте в виду, что с этими настройками ваша страница не будет отображаться во всех версиях Internet Explorer более ранних, чем IE 9. По этой причине настоящий XHTML является невыполнимым условием при заключении сделки веб-страницы с браузером.

Кстати, браузеры, поддерживающие XHTML5, обрабатывают такую разметку по-другому, чем обычный код HTML5. Они пытаются обрабатывать страницу как документ XML, и если это им не удается (по причине ошибки в коде), браузер прекращает обработку оставшейся части документа.

Какой из этого следует вывод? Для подавляющего большинства веб-разработчиков, от любителей до серьезных профессионалов, игра по строгим правилам XHTML не стоит требуемых для этого свеч. Единственным исключением является разработка специальных решений, например страниц с содержимым, которым нужно манипулировать посредством XML-инструментов, таких как, например, XQuery и XPath.

Совет

Если вам интересно, можно обмануть браузер и заставить его переключиться в режим XHTML. Для этого нужно лишь переименовать файл с расширением xhtml или xht, а потом открыть его с жесткого диска вашего компьютера. Большинство браузеров (включая Firefox, Chrome и IE 9) будут обрабатывать такую страницу, как будто бы она была загружена с веб-сервера с настройками MIME XML. Если страница содержит любую незначительную ошибку, в браузере отобразится частично обработанная страница (IE 9), сообщение об ошибке XML (Firefox) или то и другое вместе (Chrome).

Семейство элементов HTML5

Все это время мы рассматривали изменения в синтаксисе HTML5. Но более важными являются добавления, удаления и изменения поддерживаемых в HTML элементов. В последующих разделах мы вкратце рассмотрим эти аспекты.

Добавленные элементы

В следующих главах в основном внимание уделяется изучению новых элементов, предоставляющих веб-страницам возможности, которые отсутствовали до настоящего времени. Эти элементы, с кратким описанием и указанием главы, в которой они рассматриваются, перечислены в табл. 1.1.

Таблица 1.1. *Новые элементы HTML5*

Категория	Элементы	Где рассматривается
Семантические элементы для работы со структурой страниц	`<article>`, `<aside>`, `<figcaption>`, `<figure>`, `<footer>`, `<header>`, `<hgroup>`, `<nav>`, `<section>`, `<details>`, `<summary>`	Глава 2
Семантические элементы для работы с текстом	`<mark>`, `<time>`, `<wbr>` (поддерживался и ранее, но теперь официально является частью языка)	Глава 3
Элементы для работы с веб-формами и интерактивности	`<input>` (старый элемент, но со многими новыми подтипами), `<datalist>`, `<keygen>`, `<meter>`, `<progress>`, `<command>`, `<menu>`, `<output>`	Глава 4
Элементы для поддержки аудио, видео и подключаемых модулей	`<audio>`, `<video>`, `<source>`, `<embed>` (поддерживался и ранее, но теперь официально является частью языка)	Глава 5
Поддержка холста	`<canvas>`	Глава 6
Поддержка языков иных, чем английский	`<bdo>`, `<rp>`, `<rt>`, `<ruby>`	Спецификация HTML5 на сайте **http://dev.w3.org/html5/markup**

Удаленные элементы

В HTML5 были не только добавлены новые элементы, но и несколько элементов были лишены официального родства с ним.

Эти элементы будут работать в браузерах, но любой уважающий себя валидатор HTML5 *(см. разд. "Проверка кода HTML5" ранее в этой главе)* обнаружит их и закатит по этому поводу скандал.

Самыми заметными шагами при удалении были те, что поддерживают продолжаемую в HTML5 философию (впервые введенную в XHTML): *элементам оформления* не место в языке. К таким элементам относятся элементы, используемые для форматирования веб-страниц, и даже самый "зеленый" веб-разработчик знает, что это работа для таблиц стилей. Удалены, среди прочих, были элементы оформления, которые не использовались профессиональными веб-разработчиками годами, такие как `<big>`, `<center>`, `<font>`, `<tt>` и `<strike>`. Атрибуты оформления HTML постигла та же судьба, поэтому нет надобности рассматривать их здесь.

Кроме этого, HTML5 вбил осиновый кол в погребенный веб-разработчиками инструмент фреймов. Когда инструмент фреймов HTML впервые увидел свет, он казался отличным способом для отображения нескольких веб-страниц в одном окне браузера. Но в настоящее время фреймы воспринимаются больше как кошмар доступности к веб-страницам, создающим проблемы для поисковых движков, вспомогательных приложений и мобильных устройств. Но, что довольно интересно, элемент `<iframe>`, который позволяет вставлять одну страницу в другую, проскользнул в новый стандарт. Это ему удалось по той причине, что он используется в веб-страницах для выполнения ряда интеграционных задач, таких как вставка в страницы окон YouTube, рекламных блоков и поисковых полей Google.

Еще несколько элементов были удалены из-за того, что они просто предоставляли уже существующие возможности другим способом и были источником частых ошибок. Например, функцию элемента `<acronym>` лучше предоставляет элемент `<abbr>`, а элементу `<applet>` предпочтителен элемент `<object>`. Но подавляющее большинство элементов HTML было сохранено в HTML5.

> ***Примечание***
>
> Для любителей цифр, семейство HTML5 состоит немногим из более 100 элементов. Из них почти 30 новых и около 10 существенно измененных. Просмотреть список элементов (и узнать, какие из них новые или измененные) можно на веб-сайте **http://dev.w3.org/html5/markup**.

Адаптированные элементы

У HTML5 есть еще один интересный трюк — некоторые старые методы применяются с новой целью. Возьмем, например, элемент `<small>`, предназначенный для уменьшения размера шрифта в блоке текста. Будучи не самым оптимальным способом решения этой задачи, для которой лучше подходят таблицы стилей, этот элемент впал в немилость веб-разработчиков. Но в отличие от выброшенного эле-

мента <big>, элемент <small> остался в HTML5, однако функция его несколько иная. Теперь элемент <small> используется для обозначения так называемого "мелкого текста" — информации, которую не горят желанием предоставить, но которую нужно разместить согласно каким-либо требованиям. Например, предупреждение об отказе от ответственности, помещенное в самом конце контракта, наподобие следующего:

```
<small>Создатели этого сайта не несут ответственности за любые телесные
повреждения, которые пользователь может получить вследствие участия в гонках
одноколесных велосипедов без должного надзора.</small>
```

Как и ранее, текст внутри элемента <small> отображается мелким шрифтом, если только эти настройки не отменены посредством таблицы стилей.

Примечание

Мнения касательно этого нового концептуального подхода к употреблению элемента <small> разделились. С одной стороны, он хорош с точки зрения обратной совместимости, т. к. старые браузеры уже поддерживают элемент <small> и поэтому будут поддерживать его и на страницах HTML5. С другой же стороны, он вносит потенциально сбивающее с толку изменение значения для старых страниц, в которых элемент <small> применяется в оформительских целях, а не в контексте концепта "мелкого текста".

Другим элементом, чей эффект остался прежним, но концептуальное значение было модифицировано, является элемент <hr>, который рисует разделяющую линию между блоками текста. В HTML5 элемент <hr> рисует ту же самую линию, но теперь она представляет тематическое прерывание, например переход к другому вопросу. Таким образом, результат форматирования остался прежним, но имеет новое логическое значение.

Подобным образом зачеркивание шрифта элементом <s> теперь несет концептуальную нагрузку — зачеркнутый этим элементом текст считается больше неверным или утратившим значимость, вследствие чего он и был "вычеркнут" из документа. Смещение значения действия этих двух последних элементов более утонченное, чем элемента <small>, т. к. в данных изменениях перенимается способ использования этих элементов в традиционном HTML.

Полужирное и курсивное форматирование

Наиболее важными элементами с измененным значением результата их действия являются элементы для форматирования текста полужирным и курсивом. Эти два наиболее употребляемые в HTML элемента — <b> (от англ. *bold* — полужирный текст) и <i> (от англ. *italics* — курсивный текст) — были частично заменены введенными в первой версии XHTML элементами <strong> и <em>, которые выполняли то же самое форматирование. Идея заключалась в том, чтобы перестать рассматривать эти элементы с точки зрения простого форматирования, а придать им логический смысл. Таким образом, элемент <strong> должен был применяться для логического выделения полужирным форматированием текста, имеющего важное (strong) значение, а элемент <em> для акцентирования (stress emphasis) курсивом определен-

ного блока текста. Идея была довольно рациональной, но элементы <b> и <i> продолжали употребляться как более короткие и знакомые заменители введенных в XHTML новшеств.

В HTML5 предприняли другую попытку решить этот вопрос. Вместо того чтобы насильно отваживать разработчиков от использования элементов <b> и <i>, результатам действия этих элементов добавили новые логические значения. Целью является позволить всем четырем элементам мирно сосуществовать в солидном документе HTML5. В результате получился следующий, несколько размытый, набор правил.

- Элемент <strong> следует использовать для выделения текста, имеющего *важное значение*, чтобы выделить его из остального текста.
- А элемент <b> следует использовать для текста, который не более важен, чем остальной текст, но по каким-либо причинам должен выделяться из него. Таким текстом, среди прочего, могут быть ключевые слова, названия товаров и все прочее, что было бы выделено полужирным в традиционной печати.
- Аналогично элемент <em> следует использовать для акцентирования текста, на который *делается ударение*; иными словами, текста, который произносился бы с ударением в устной речи.
- А элемент <i> используется для форматирования текста, который по той или иной причине нужно выделить курсивом, но на который не делается никакого особого ударения. Это могут быть, среди прочего, иностранные слова, технические термины и любой текст, который бы выделялся курсивом в печати.

В следующем листинге приводится пример разметки, в которой все эти четыре элемента используются соответствующим образом:

```
<strong>Экстренное сообщение!</strong> В кондитерской <b>Ажурная</b> идет
распродажа конфет <i>Птичье молоко</i>. Не мешкайте, потому что с уходом
последней конфеты такой возможности больше не представится <em>никогда</em>.
```

В браузере этот текст будет выглядеть таким образом:

> **Экстренное сообщение!** В кондитерской **Ажурная** идет распродажа конфет *Птичье молоко*. Не мешкайте, потому что с уходом последней конфеты такой возможности больше не представится *никогда*.

Остается лишь ждать, будут ли веб-разработчики следовать благим намерениям HTML5 или же предпочтут использовать наиболее знакомые элементы для полужирного и курсивного форматирования.

Подкорректированные элементы

В HTML5 также изменены правила для нескольких элементов. Обычно эти изменения затрагивают только незначительные аспекты, которые заметят лишь фанаты HTML, но иногда они имеют более глубокий эффект. В качестве примера, среди прочих, можно взять редко используемый элемент <address>, который, несмотря на свое название, не подходит для работы с почтовыми адресами. Его назначение бо-

лее узкое — предоставление контактной информации о создателе документа HTML, обычно адреса электронной почты или ссылки на веб-сайт, как показано в следующем примере:

```
Нашим веб-сайтом управляют:
<address>
<a href="mailto:jsolo@mysite.com">John Solo</a>,
<a href="mailto:icheng@mysite.com">Lisa Cheng</a>, and
<a href="mailto:rpavane@mysite.com">Ryan Pavane</a>.
</address>
```

Также несколько изменилось назначение элемента `<cite>`. Его можно продолжать использовать для указания названия какой-либо интеллектуальной работы, например рассказа, статьи, телевизионной программы и т. п., следующим образом:

```
<p>Чарльз Диккенс написал <cite>Повесть о двух городах</cite>.</p>
```

Но элемент `<cite>` больше неприемлемо использовать, чтобы помечать имена людей. Это ограничение оказалось на удивление дискуссионным, т. к. такое применение раньше позволялось. Некоторые веб-разработчики уровня гуру публично призывают разработчиков не обращать внимания на это ограничение. Такое внимание довольно странно, т. к. если бы вы были редактором веб-страниц, то могли бы заниматься этим всю жизнь, и никогда не встретить элемент `<cite>`.

Более серьезной корректировке подвергся элемент `<a>`, используемый для создания ссылок. В более ранних версиях HTML элементу `<a>` разрешалось содержать текст или изображение со ссылкой. А в HTML5 он может содержать все и вся, что означает, что теперь ссылки можно делать из целых абзацев текста, вместе со списками, изображениями и т. п. Текст со ссылкой подчеркивается, а шрифт окрашивается в синий цвет, как и границы изображения. Веб-браузеры поддерживали эти свойства годами, но только в HTML5 они приобрели официальный статус стандарта, что, впрочем, не делает их более или особенно полезными.

Некоторые корректировки еще не работают ни в одном браузере. Например, элемент `<ol>` (от англ. *ordered list* — упорядоченный список) теперь имеет атрибут `reversed` (реверс), который позволяет вести счет в обратном направлении (или к единице, или к любому другому значению, указанному в атрибуте `start`). Но эта возможность еще не распознается ни одним браузером.

В этой книге мы рассмотрим еще несколько подкорректированных подобным образом элементов.

Стандартизированные элементы

В HTML5 также добавлена официальная поддержка нескольких элементов, которые поддерживались в языках HTML и XHTML, но на неофициальном уровне. Одним из лучших примеров таких элементов будет элемент `<embed>`, который используется везде и повсюду в Паутине как универсальный метод для втискивания в страницу подключаемого модуля.

Более экзотичным является элемент <wbr>>, который указывает необязательный разрыв слова, т. е. место, в котором браузер может сделать перенос текста на новую строку, если слово не помещается в контейнер:

```
<p>Many linguists remain unconvinced that
<b>supercali<wbr>fragilistic<wbr>expialidocious</b> is indeed a word.</p>
```

Элемент <wbr> полезен, когда нужно вставить длинные названия (которые иногда встречаются в терминологии программирования) в ограниченное пространство, например в ячейку таблицы или текстовое поле. Но даже если браузер поддерживает элемент <wbr>, он разорвет слово только в том случае, если оно не помещается в доступное пространство. Таким образом, слово в предыдущем примере браузер может отобразить одним из следующих образов, представленных на рис. 1.4.

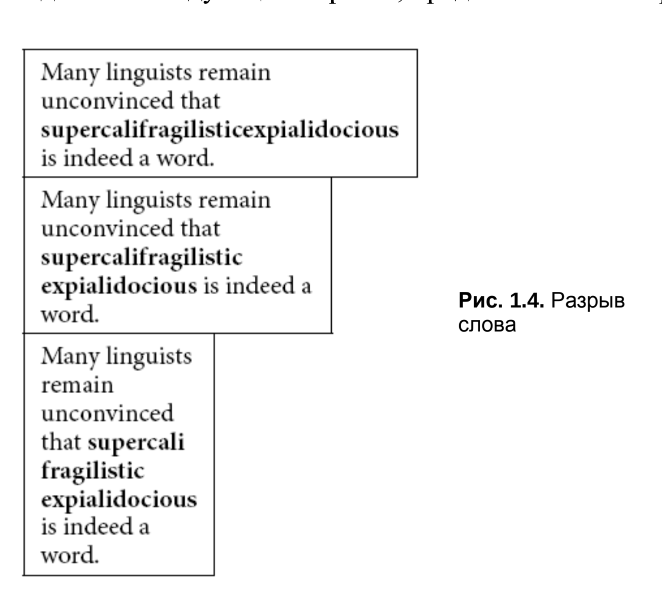


Рис. 1.4. Разрыв слова

Элемент <wbr является естественным дополнением к стандартному элементу <nobr>, который используется для предотвращения разрыва слова, когда для него имеется достаточно места. Но вплоть до HTML5 он не поддерживался никаким стандартом и распознавался только некоторыми браузерами.

Современное использование HTML5

Как можно видеть, стандарт HTML5 поддерживает всевозможные странные методы. В то же самое время он возрождает (и стандартизует) некоторые старые либеральные правила HTML и вводит передовые возможности, которые работают только в новейших браузерах.

Что касается браузерной совместимости, функциональные возможности HTML5 можно разбить на три категории.

- **Возможности, которые уже работают.** К этой категории принадлежат возможности, которые имеют высокий уровень поддержки, но официально не были

частью стандарта HTML в прошлом (например, метод drag and drop). В нее также входят возможности, которые можно реализовать для старых браузеров, приложив очень небольшие дополнительные усилия, наподобие семантических элементов, рассматриваемых в *главе 2*.

- Возможности с поэтапной деградацией. Например, если у старого браузера имеются проблемы с использованием нового элемента `<video>`, этот элемент позволит вам предоставить браузеру какое-либо другое средство проигрывания, например плеер, использующий подключаемый модуль Flash. Это намного лучше, чем сообщение об ошибке, которое определенно нельзя назвать поэтапной деградацией. В эту категорию также входят второстепенные возможности из разряда примочек, на которые старые браузеры могут спокойно не обращать внимания, наподобие некоторых старых возможностей для работы с веб-формами (например, замещающий текст), и некоторые возможности форматирования из CSS3 (например, для скругления углов или отбрасывания теней).
- Возможности, требующие обходных решений JavaScript. Мотивацией для многих новых возможностей HTML5 послужили разработки, которые веб-программисты уже делали другим, более трудоемким, способом. Поэтому не стоит удивляться, что много из возможностей HTML5 можно дублировать с помощью хорошей библиотеки JavaScript (или, в худшем случае, написав энное количество строк кода собственного специализированного сценария JavaScript). Создание обходного решения JavaScript может быть очень трудоемкой задачей, поэтому, если вы посчитаете, что определенная возможность является необходимой и требует обходного решения JavaScript, вы можете просто решить использовать это обходное решение для всех страниц и отложить использование методов HTML5 на будущее.

В этой книге предполагается, что предпочтения будут отданы подходу с использованием методов HTML5 в случаях, когда это абсолютно безопасно. Это будут методы из первой категории вышеизложенного списка. Если возможность принадлежит одной из двух других категорий (что бывает достаточно часто), то вам нужно решить, стоит ли погружаться в разработку решения, работающего в старых браузерах, или же будет предпочтительней подождать и подготовиться к лучшей ситуации в будущем. Если в Паутине есть хорошее обходное решение на JavaScript, то оно будет указано в этой книге. Но во многих случаях вам придется делать выбор между замысловатой новой возможностью и универсальным доступом.

Поддерживает ли браузер вашу разметку?

Последнее слово в вопросе, в каком объеме использовать HTML5, принадлежит разработчикам браузеров. Если браузер не поддерживает какую-либо возможность, нет никакого смысла использовать ее, несмотря на все, что говорится в стандарте. В настоящее время существуют четыре или пять основных браузеров (не считая браузеров для мобильных устройств с возможностью подключения к Интернету, таких как смартфоны и планшеты iPad). У разработчика-одиночки нет никаких шансов протестировать каждую потенциальную возможность на каждом браузере,

не говоря уже о возможности оценить поддержку ее в старых версиях браузеров, которые широко используются до сих пор.

К счастью, существует веб-сайт **http://caniuse.com**, который может помочь вам справиться с этой задачей. В нем даются подробности поддержки HTML5 во *всех* основных браузерах. И, самое приятное, он позволит вам выделить именно те возможности, которые вам требуются. Веб-сайт работает следующим образом:

1. Сначала выберите вкладку **Tables**, а потом вкладку **Compatibility tables** и выберите на ней требуемую вам возможность (или группу возможностей), установив соответствующие флажки (рис. 1.5)[1].

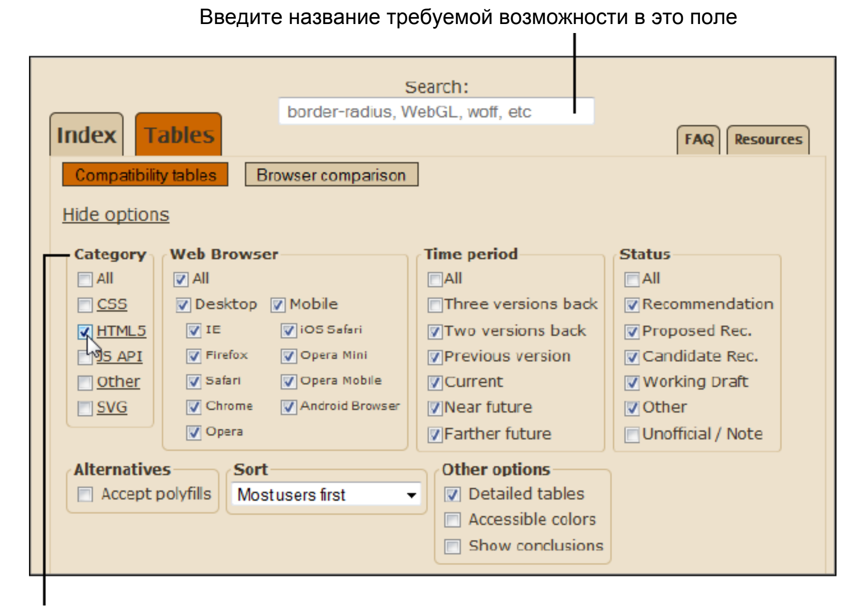


Рис. 1.5. Поиск по заданным параметрам проверяет поддержку только основных возможностей HTML5, но для всех версий всех браузеров

Можно выполнить поиск конкретной возможности, введя ее название в поле **Search**, расположенное по центру вверху страницы. Или же можно просмотреть целую категорию возможностей, установив соответствующий флажок в разделе **Caterogy** слева на странице. (В таком случае будет выведена таблица совместимости для каждой вложенной возможности.) Например, чтобы проверить только возможности, которые считаются частью стандарта HTML5, сбросьте все флаж-

[1] Интерфейс веб-сайтов меняется со временем. На рис. 1.5 представлен вид сайта на момент написания книги. Сейчас эта веб-страница выглядит иначе. Чтобы увидеть флажки, надо щелкнуть по ссылке **Show options** в левой части веб-страницы. (На рис. 1.5 эта ссылка после щелчка называется **Hide options**.) Вполне возможно, читатель увидит иной интерфейс сайта после выхода книги из печати. — *Ред.*

ки и установите только флажок **HTML5**. Чтобы проверить совместимость возможностей на основе JavaScript, которые сначала входили в HTML5, но потом были выделены в отдельную категорию, установите флажок **JS API**. А для проверки поддержки новых искусных возможностей CSS3 установите флажок **CSS**.

2. При желании, выберите другие опции, установив соответствующие флажки.

   Можно уточнить результаты поиска, удалив некоторые подробности. Например, возможно, вас не интересует информация о совместимости с браузерами для мобильных устройств или с браузерами, которые находятся в стадии разработки и не были официально выпущены. Но обычно не стоит отказываться от этих подробностей, т. к. таблицы легко понимать даже с ними.

3. Прокрутите страницу вниз, чтобы просмотреть все результаты (рис. 1.6).



Рис. 1.6. В настоящее время результаты поддержки браузерами всех возможностей HTML5 отображаются в 20 таблицах. В таблице на рисунке отображается состояние поддержки браузерами HTML5-элемента `<audio>`

Для большого количества возможностей одновременно выводится только 20 таблиц результатов. Для просмотра следующих 20 таблиц результатов следует щелкнуть по ссылке **Show next 20** внизу страницы.

В таблице для каждой возможности в заголовках столбцов указаны браузеры, а в заголовках строк — их характеристики версий. Определенная версия браузера

находится на пересечении соответствующего столбца и строки. Если возможность поддерживается данной версией браузера, соответствующая ячейка закрашена светло-зеленым цветом; частичная поддержка обозначается темно-зеленым, а отсутствие поддержки — розовым. Если неизвестно, поддерживается ли данная возможность, в ячейке не указывается номер версии браузера, а сама ячейка окрашена коричневым цветом.

Статистика популярности браузеров

Последним важным пунктом проблемы поддержки возможностей браузерами является статистика популярности конкретных браузеров. Иными словами, информация о том, сколько посетителей Паутины пользуется браузером, поддерживающим возможности, которые вы намереваетесь использовать в своей разметке. Одним из хороших источников этой информации является популярный сайт GlobalStats (**http://gs.statcounter.com**). На странице сайта в раскрывающемся списке **Statistic** выберите вариант **Browser**. А вариант **Browser Version** позволит просмотреть популярность не только конкретного браузера, но и каждой из его версий. Результаты можно сузить, выбрав конкретный регион или страну в раскрывающемся списке **Country/Region**.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Internet Explorer утрачивает популярность?

В процессе изучения HTML5 вы быстро обнаружите, что один браузер выделяется на фоне всех остальных своим самым низким уровнем поддержки HTML5 (а также самой медленной скоростью выхода обновлений). Этим плетущимся в хвосте браузером является Internet Explorer. Хотя выпуском IE 9 корпорация Microsoft сделала громадный шаг в правильном направлении, уровень поддержки HTML5 в IE все еще ниже, чем в любом другом современном браузере.

Не то чтобы Microsoft не обращает внимания на последние разработки в области HTML5, но она удерживается от реализации возможностей, которые еще не завершены. Для некоторых из этих возможностей Microsoft предоставляет экспериментальные расширения к IE (называющиеся "лабораториями" HTML5), которые можно загрузить с сайта **http://html5labs.interoperabilitybridges.com/**. Все это хорошо, если вы просто хотите немного поиграть с некоторыми новыми возможностями HTML5, которые все еще находятся в процессе разработки. Но если вы хотите реализовать эти возможности в настоящем веб-приложении, то помощи от этого мало. (Конечно же, реализация некоторых из этих возможностей граничит с безрассудством, но это уже совершенно другой вопрос.)

Более того, с поддержкой HTML5 в IT имеется еще более важная проблема. Как уже упоминалось, IE 9 является первой версией IE, поддерживающей возможности HTML5. Но IE 9 устанавливается только под Windows 7 или Vista. Пользователи с Windows XP (которая, несмотря на свой почтенный возраст, на момент написания этой книги была самой популярной операционной системой в мире и по всем признакам будет широко использоваться еще долгое время, даже после того, как это место займет Window 7), не могут установить IE 9 и, соответственно, не могут получить *никакой* поддержки для HTML5. И хотя эти оставленные на произвол судьбы компанией Microsoft компьютеры не полностью потеряны — на них можно установить какой-либо другой браузер — пройдет очень много времени, прежде чем веб-разработчики смогут перейти на HTML5, не беспокоясь об обнаружении возможностей и резервных решениях.

Наконец, с помощью переключателей справа от области вывода результатов можно выбрать способ отображения результатов. Столбиковая диаграмма отображает популярность браузеров в настоящее время. Для ее вывода отметьте переключатель **Bar**. А линейный график показывает тенденции в выборе браузера в течение определенного периода времени. Этот режим устанавливается переключателем **Line**. По умолчанию в режиме линейного графика отображаются результаты за целый год, начиная от текущего месяца (рис. 1.7). Но период времени можно настроить, выбрав соответствующее значение в поле **Time Period**.

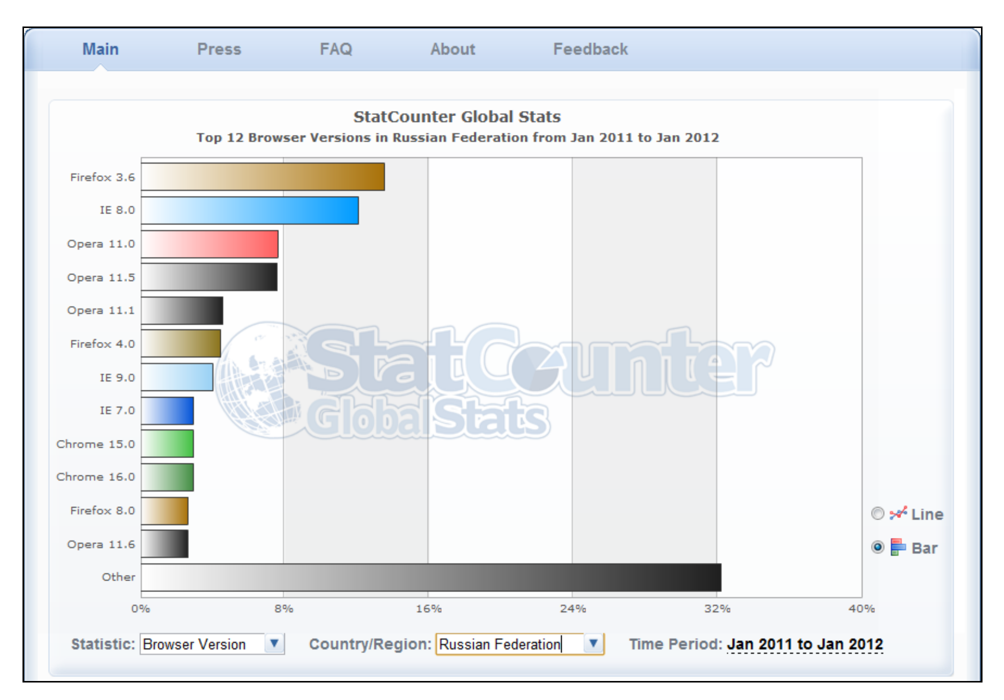


Рис. 1.7. Как видно на рисунке, поддержка HTML5 браузерами не впечатляет

Сайт GlobalStats собирает статистические данные ежедневно с помощью кода слежения, который установлен на миллионах веб-сайтов. Но в то время как эти статистические данные представляют огромное количество страниц и громадный объем данных, все равно это лишь малая часть всей Паутины. Это означает, что нельзя непременно полагать, что посетители вашего веб-сайта будут пользоваться такими же браузерами.

Более того, браузерные предпочтения могут быть разными в зависимости от страны пользователя и типа веб-сайта. Например, в Германии наиболее популярным браузером является Firefox, которому отдают предпочтение 60% пользователей веб, а в Белоруссии это Opera с 49% доли пользователей Интернета. А на веб-сайт TechCrunch (популярный новостной сайт для компьютерных фанатов) всего лишь 16% посетителей заходят с помощью самого популярного в мире браузера, Internet Explorer. Поэтому, если вы хотите разработать веб-сайт для пользователей своего круга, изу-

чите статистические данные для страниц своего веб-сайта. (Если вы еще не пользуетесь услугами сбора статистических данных для своего сайта, вас может заинтересовать совершенно бесплатная услуга высокого уровня Google Analytics. См. **www. google.com/analytics**.)

Определение возможностей с помощью Modernizr

В течение нескольких следующих лет абсолютно реальным будет то обстоятельство, что некоторые посетители вашего веб-сайта будут пользоваться браузерами, не поддерживающими HTML5. Но это не должно остановить вас от использования возможностей этого стандарта при условии, что вы согласны вложить немного усилий в разработку обходных решений или возможности поэтапной деградации. В любом случае вам, скорее всего, потребуется некоторая помощь JavaScript. Обычно это делается таким образом: после загрузки вашей страницы браузером запускается специальный сценарий, который проверяет, поддерживает ли браузер определенную возможность.

К сожалению, т. к. в своей основе HTML5 является набором связанных стандартов, одного общего теста на поддержку возможностей не существует. Вместо этого требуется выполнить несколько десятков разных тестов, чтобы проверить наличие поддержки различных возможностей, при этом иногда даже требуется проверять, поддерживается ли определенная часть возможности, что очень быстро делает задачу тестирования весьма неприятной.

При проверке поддержки некоторой функциональности браузером обычно требуется найти возможность в программном объекте или создать объект и использовать его определенным образом. Но подумайте хорошенько, прежде чем приступать к написанию кода тестирования такого типа, т. к. с этой задачей очень легко наломать дров. Например, код для проверки поддержки возможности браузерами может не работать на некоторых браузерах по той или иной непонятной причине или быстро устареть. Вместо этого подумайте о применении компактного, постоянно обновляемого инструмента Modernizr (**www.modernizr.com**), который проверяет поддержку браузерами широкого диапазона возможностей HTML5 и связанных возможностей. Он также предоставляет классный метод для реализации поддержки резервных решений при использовании новых возможностей CSS3 *(см. разд. "Стратегия 3: добавляйте резервные решения с помощью Modernizr" главы 8)*.

Проверка веб-страниц с помощью Modernizr выполняется так:

1. Загрузите JavaScript-файл для Modernizr. Для этого в области **Download Modernizr** в верхней центральной части страницы нажмите кнопку **Development**.

   Обычно, имя этого файла будет похоже на modernizr-2.0.6.js. Точное имя зависит от используемой версии. Некоторые разработчики переименовывают этот файл, убирая номер версии, например modernizr.js. Это позволяет обновить файл Modernizr в будущем, не требуя поиска и корректировки ссылок на него в веб-страницах, в которых он используется.

Совет

Код полной версии Modernizr несколько объемистый. Эта версия сценария предназначена для выполнения тестирования на стадии разработки веб-сайта. По окончании разработки, когда сайт можно публиковать для использования, можно создать облегченную версию сценария Modernizr, которая тестирует только требуемые возможности. Для этого загрузите версию Production, нажав одноименную кнопку в области **Download Modernizr**. Откроется страница, предлагающая выбрать возможности, поддержку которых вы хотите проверять (установив соответствующие флажки), а потом создать свою специальную версию сценария Modernizr, нажав кнопку **Generate** слева внизу страницы.

2. Скопируйте файл сценария в папку, в которой находится веб-страница, требующая тестирования. Либо файл сценария можно поместить в подкаталог и подкорректировать должным образом путь к ней в ссылке JavaScript (см. следующий шаг).
3. В блоке <head> тестируемой веб-страницы вставьте ссылку на файл сценария Modernizr. В следующем листинге показан пример вставки этой ссылки:

```
<head>
  <meta charset="utf-8">
  <title>HTML5 Feature Detection</title>
  <script src="modernizr-2.0.6.js"></script>
  ...
</head>
```

Теперь, всякий раз при загрузке этой страницы будет исполняться сценарий Modernizr. В считанные миллисекунды сценарий тестирует поддержку пары десятков новых возможностей, а потом создает объект JavaScript, называющийся, опять же, Modernizr и содержащий результаты тестирования. Чтобы проверить поддержку браузером определенной возможности, тестируются свойства этого объекта.

Совет

Полный список тестируемых с помощью Modernizr возможностей, а также код JavaScript для тестирования каждой из этих возможностей, см. в документации по ссылке **www.modernizr.com/docs**.

4. Напишите сценарий, который тестирует требуемую возможность, а потом выполняет соответствующее действие. Пример возможного сценария для проверки, поддерживается ли HTML5-возможность drag and drop, и вывода результатов в окне браузера показан в следующем листинге:

```
<!DOCTYPE  html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>HTMI_5 Feature Detection</title>
  <script src="modernizr-2.0.6.js"></script>
</head>

<body>
  <p>The verdict is... <span id = "result"x/span></p>
```

```
  <script>
    // Найти элемент на странице (называющейся result), в котором
    // можно вывести результаты.
    var result = document.getElementById("result");
    if (Modernizr.draganddrop) {
        result.innerHTML = "Rejoice! Your browser supports" +
                           "drag-and-drop.".
    }
    else {
        result.innerHTML = "Your feeble browser doesn't " +
                           "support drag-and-drop.";
    }
  </script>
</body>
</html>
```

Результаты исполнения этого сценария показаны на рис. 1.8.

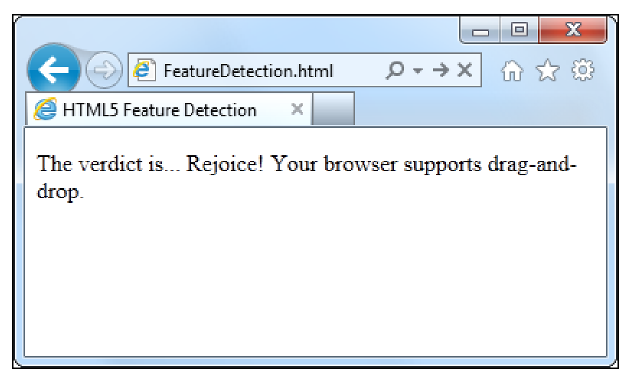


Рис. 1.8. Результат исполнения сценария тестирования поддержки браузером возможности drag and drop

Хотя в этом примере показан правильный способ проверки поддержки возможности, применяемый в нем подход к обработке неподдерживаемой возможности не идеален. Вместо того чтобы просто проинформировать (пусть даже и самым вежливым образом) посетителя вашего веб-сайта о том, что его браузер не поддерживает определенную функциональность вашего сайта, намного лучше будет реализовать какое-либо обходное решение, даже если это решение и не будет таким изящным или обладать всеми способностями заменяемой возможности HTML5. Например, если неподдерживаемая возможность — всего лишь какая-то несущественная примочка, которая бесполезна для посетителя сайта, то эту проблему можно вообще игнорировать.

Совет

В этом примере применяются базовые, проверенные временем, методы JavaScript — поиск элемента по его ID и изменение содержимого элемента. Если вы не вполне понимаете, как это работает, обратитесь к *приложению 2*.

Замена отсутствующих возможностей заполнителями

Инструмент Modernizr позволяет разработчику обнаружить дыры в поддержке возможностей браузером, информируя его о неработающей функциональности. Но он ничего не делает для решения этих проблем. И здесь на выручку приходят заполнители[1] (polyfills, полифилы). По сути, заполнители — это разношерстный набор методов для заполнения дыр в поддержке старыми браузерами стандарта HTML5. Английское слово *polyfill* происходит от продукта *polyfiller*, который представляет собой смесь для заделки дыр в гипсокартоне перед покраской, т. е. попросту вид шпаклевки. В HTML5 идеальным заполнителем будет такой, который можно вставить в страницу без излишних усилий. Такой заполнитель предоставляет обратную совместимость без прерываний в работе страницы и не влияет на остальной код страницы, что позволяет разработчику работать с чистым HTML5, в то время как кто-то другой беспокоится об обходных решениях.

Но заполнители — не совсем идеальное решение. Некоторые из них полагаются на другие технологии, для которых может предоставляться только частичная поддержка. Например, один из полифилов позволяет эмулировать элемент HTML5 `<canvas>` на старых версиях Internet Explorer с помощью подключаемого модуля Silverlight. Это хорошо, если на браузере посетителя вашей веб-страницы установлен данный модуль. Но если нет, и посетитель не хочет или не может установить его, то вам нужно прибегнуть к какому-либо другому решению. Другие заполнители могут быть менее работоспособными, чем настоящая возможность HTML5, или менее производительными.

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ
Модифицирование IE с помощью Google Chrome Frame

Самым большим изо всех заполнителей является Google Chrome Frame, подключаемый модуль браузера для Internet Explorer версий 6, 7, 8 и 9. Этот заполнитель запрашивает исполнение браузера Chrome в процессе работы IE и обрабатывает страницы HTML5. Но Chrome Frame запускается не для всех HTML5-страниц, а только для таких, в которые их разработчик вставил информацию, явно разрешающую это сделать.

Очевидным ограничением Chrome Frame является то обстоятельство, что для того, чтобы веб-сайты могли использовать его, пользователи должны установить его на свои браузеры. А если они не имеют ничего против этого, то почему бы тогда не установить полноценный браузер Chrome? В любом случае, если вы хотите узнать больше о Google Chrome Frame, то можете обратиться по адресу **http://code.google.com/chrome/chromeframe**.

В этой книге дается информация о некоторых возможных заполнителях. Более подробную информацию можете найти самое что ни на есть близкое к всеохватывающему каталогу полифилов HTML5 на веб-сайте GitHub (**https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills**). Но имейте в виду — диапазон качества, производительности и поддержки заполнителей очень большой.

[1] Устоявшегося русского термина пока нет. Поэтому назовем это средство словом, выражающим смысл данного средства. — *Ред.*

Совет

Помните, что недостаточно просто знать, что для данной HTML5-возможности существует заполнитель. Необходимо протестировать его и проверить качество его работы на разных старых браузерах, прежде чем идти на риск и публиковать соответствующую возможность на своем веб-сайте.

Имея в своем распоряжении инструменты для сбора статистических данных о браузерах, тестирования функциональностей и создания обходных решений, вы готовы к серьезному рассмотрению использования возможностей HTML5 в своих веб-страницах. В следующей главе мы сделаем первый шаг к этому, рассмотрев некоторые элементы HTML5, которые работают как в новых, так и в старых браузерах.

2

ГЛАВА 2

Новый способ структурирования страниц

За почти два десятилетия существования Интернета веб-сайты претерпели разительные изменения. Но больше всего удивляют не масштабы изменения Интернета, а то, как хорошо сохранились древние элементы HTML. По существу, веб-разработчики создают современные веб-сайты, используя такой же набор элементов, что и десять лет тому назад.

Один элемент в особенности — скромный `<div>` (от англ. *division* — раздел, блок) — является краеугольным камнем каждой современной веб-страницы. С помощью элементов `<div>` в документе HTML можно разместить колонтитулы, боковые панели, панели навигации и многое другое. Добавив хорошую щепотку CSS-форматирования, эти секции можно превратить в блоки с границами или затененные колонки и поместить каждый из них точно в требуемом месте.

Форматирование страниц с применением элемента `<div>` и таблиц стилей — метод прямолинейный, мощный и гибкий. Но не *прозрачный*. Это означает, что изучение разметки другого разработчика требует определенных усилий в том, чтобы разобраться в каждом элементе `<div>` и целиком во всей странице. Чтобы понять логику разметки, необходимо перескакивать туда и обратно между разметкой, таблицами стилей и отображением страницы в браузере. С таким затруднением вам придется столкнуться при рассмотрении более-менее сложной веб-страницы любого разработчика, хотя, скорее всего, вы применяете такие же методы для создания своих веб-страниц.

Эта ситуация заставила разработчиков задуматься, нельзя ли заменить элемент `<div>` чем-либо лучшим? Чем-то, что работало бы подобно `<div>`, но было бы более осмысленным. Чем-то, что помогло бы отделить боковые панели от заголовков, а рекламные панели от меню. Стандарт HTML5 позволяет сделать эту мечту реальностью, предоставляя набор новых элементов для структурирования страниц.

Совет

Если ваших знаний CSS недостаточно, чтобы понять, о чем идет речь в какой-либо рассматриваемой таблице стилей, тогда вы не совсем готовы для изучения этой главы. Но не отчаивайтесь, в *приложении 1* представлено введение в основы CSS, знакомства с которым будет достаточно для продолжения изучения HTML5.

Что такое семантические элементы?

Новые *семантические элементы* HTML5 позволяют улучшить структуру веб-страницы, добавляя смысловое значение заключенному в них содержимому.

Например, новый элемент `<time>` обозначает действительную дату или время веб-страницы. Вот самый простой пример использования этого элемента:

```
Регистрация начнется <time>2012-11-25</time>.
```

В браузере эта разметка отображается таким образом:

```
Регистрация начнется 2012-11-25.
```

Здесь важно понимать то обстоятельство, что элемент `<time>` не обладает никакими встроенными возможностями форматирования. По сути, ничто не говорит устройству для чтения веб-страниц о том, что дата в коде страницы заключена в дополнительный элемент. Хотя к элементу `<time>` можно добавить дополнительное форматирование с помощью таблицы стилей, по умолчанию он ничем не отличается от обычного текста.

В данном случае, элемент `<time>` предназначен содержать одну единицу информации. Но большинство семантических элементов HTML5 не такие: они служат для обозначения блоков содержимого большего размера. Например, элемент `<nav>` обозначает набор ссылок для перемещения по содержимому, элемент `<footer>` в коде обрамляет нижний колонтитул страницы и т. д. для десятка (или около этого) новых элементов.

> **Примечание**
>
> Хотя семантические элементы наименее впечатляющие из новых возможностей HTML5, они составляют одну из самых больших групп. По сути, большинство новых элементов HTML5 является семантическими элементами.

Все семантические элементы имеют одну общую отличительную особенность: они по существу ничего не делают. В противоположность, элемент `<video>`, например, вставляет в веб-страницу полноценный видеоплеер *(см. разд. "Воспроизведение видео с помощью элемента <video>" главы 5)*. Зачем же тогда утруждать себя использованием набора новых элементов, которые никак не изменяют внешний вид веб-страницы?

Этому есть несколько хороших причин.

- **Более удобное редактирование и сопровождение.** Разметка традиционной HTML-страницы может быть трудной для понимания. Чтобы понять общую структуру и значение отдельных блоков страницы, часто приходится исследовать ее таблицу стилей. А использование семантических элементов HTML5 позволяет разработчику предоставить в разметке страницы дополнительную информацию о ее структуре. Это облегчит вам жизнь, когда потребуется редактировать эту страницу через несколько месяцев, и будет еще более кстати, если вашу работу придется редактировать кому-то другому.
- **Доступность.** Одной из ключевых тем в сфере современного веб-дизайна является создание доступных страниц, т. е. страниц, которые пользователи могут

просматривать и перемещаться по ним с помощью программ-ридеров[1] и других вспомогательных средств. В настоящее время разработчики инструментов, делающих контент страницы доступным, все еще пытаются догнать HTML5. Когда они, наконец, достигнут весомого результата, то смогут предоставить намного лучшие возможности просмотра веб-страниц для пользователей с ограниченными возможностями. (Только один пример, представьте себе, как программа чтения экрана (скрин-ридер) может использовать элементы `<nav>`, чтобы быстро найти ссылки для перемещения по странице или на внешние ресурсы.)

СОВЕТ

С дополнительной информацией о передовых практиках в области доступности веб-страниц можно ознакомиться на веб-сайте организации WAI[2] (**www.w3.org/WAI**). Или же, чтобы получить представление о жизни со считывателем экрана (и заодно узнать, почему так важно правильное размещение заголовков), можете просмотреть видео на YouTube (**http://tinyurl.com/6bu4pe**).

- **Оптимизация поисковых движков.** Поисковые движки наподобие Google используют мощные *поисковые роботы* — автоматизированные программы, которые методически обходят Интернет и просматривают все страницы, которые они могут найти — для сканирования содержимого веб-страниц и составления для них указателей в своих поисковых базах данных. Чем лучше Google понимает вашу веб-страницу, тем больше шансов, что он сможет сопоставить ее содержимое с поисковым запросом, и тем больше шансов, что ваша веб-страница будет отображена в результатах чьего-либо поиска. В настоящее время поисковые роботы уже проверяют на наличие некоторых семантических элементов HTML5, чтобы собрать всю возможную информацию об индексирующих их веб-страницах.
- **Поддержка будущих возможностей.** В новых браузерах и инструментах редактирования веб-страниц несомненно будет использоваться самыми разнообразными способами весь диапазон предоставляемых семантическими элементами возможностей. Например, браузер может предоставлять посетителям веб-страницы сводку ее содержимого, что позволит им быстро перейти к интересующему их разделу страницы, наподобие **Панели навигации** в Microsoft Word 2010. (Надо сказать, что браузер Chrome уже имеет подключаемый модуль, который как раз это и делает, см. рис. 2.10.) Подобным образом инструменты для веб-разработки могут содержать возможности, позволяющие создавать или редактировать меню навигации посредством управления содержимым, помещаемым в блок `<nav>`.

Из всего этого можно сделать следующий вывод: правильно используя семантические элементы, можно создавать более аккуратные и понятные веб-страницы, готовые для работы со следующим поколением браузеров и веб-приложений. Но если ваше мышление все еще сковано устаревшим подходом традиционного HTML, будущее может просто обойти вас стороной.

---

[1] Их называют скрин-ридерами — от англ. *screen reader*. — *Ред.*

[2] Web Accessibility Initiative — Инициатива по общедоступности сети.

Модифицирование традиционной HTML-страницы

Легче всего начать знакомство с новыми семантическими элементами, а также обучаться их применению для структурирования страницы на классическом HTML-документе и вставить в него некоторые облагоображивающие элементы HTML5. На рис. 2.1 показан первый пример, который вы можете попробовать реализовать.

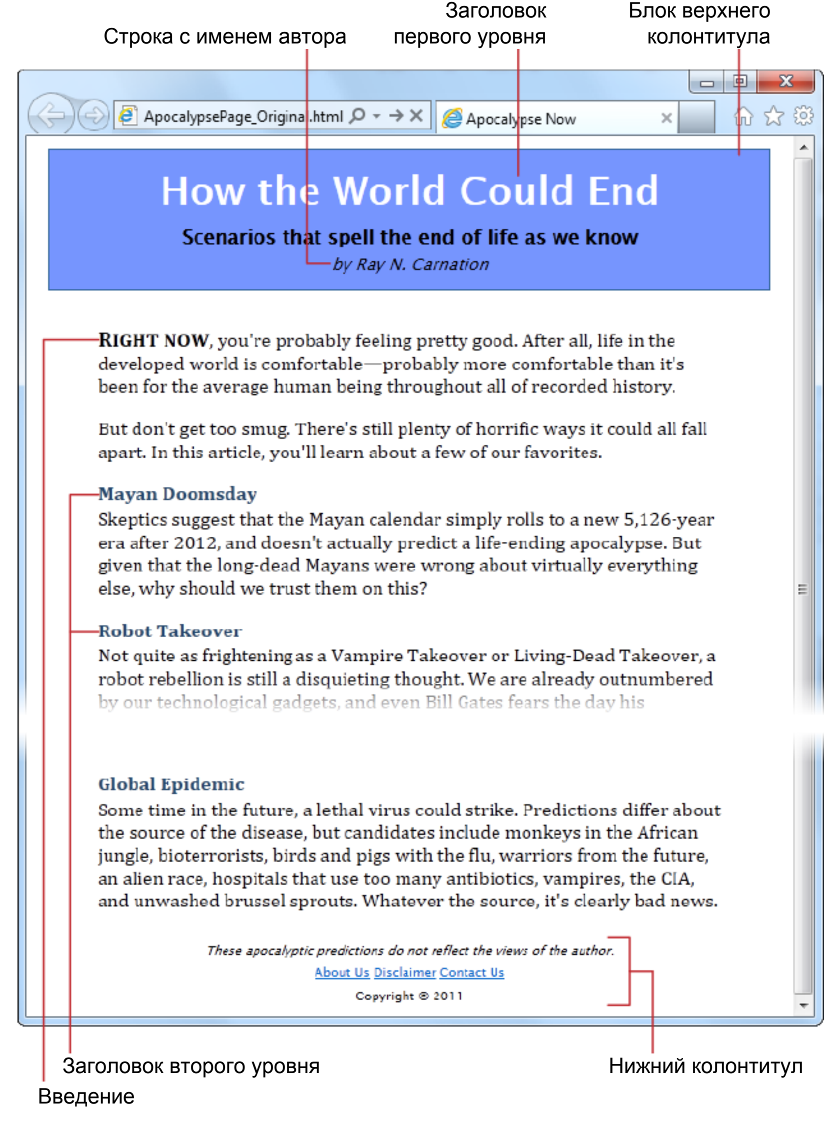


Рис. 2.1. Эта HTML-страница имеет базовую структуру, подобную структуре документа. Все форматирование осуществляется с помощью связанной таблицы стилей

Это простая, самодостаточная веб-страница, содержащая статью, хотя другие типы содержимого (например, сообщения блогов, описание продукта или короткий рассказ) вполне бы подошли для этих целей.

> ***Совет***
>
> Пример на рис. 2.1, а также все другие примеры в этой книге можно просмотреть или загрузить из сайта книги по адресу **http://www.prosetech.com/html5/**. Если вы хотите модифицировать HTML-код сами, начните со страницы ApocalypsePage_Origmal.html, а если желаете сразу же увидеть конечный HTML5-результат, просмотрите страницу ApocalypsePage_Revised.html.

Структурирование страницы старым способом

Отформатировать страницу, показанную на рис. 2.1, можно несколькими разными способами. К счастью, в этом примере применены самые передовые подходы HTML, поэтому в разметке нет ни намека на средства форматирования. В ней нет ни элементов для полужирного или курсивного начертания, ни вставляемых в строку разметки стилей и определенно ничего похожего на такое уродство, как устаревший элемент `<font>`. Вместо этого документ аккуратно отформатирован посредством связи с внешней таблицей стилей.

В следующем листинге приводится сокращенная версия разметки. В ней жирным шрифтом выделены места, в которых документ привязывается к таблице стилей:

```
<div class="Header">
  <h1>How the World Could End</h1>
  <p class="Teaser">Scenarios that spell the end of life as we know</p>
  <p class="Byline">by Ray N. Carnation</p>
</div>

<div class="Content">
  <p><span class="LeadIn">Right now</span>, you're probably...</p>
  <p>...</p>

  <h2>Mayan Doomsday</h2>
  <p>Skeptics suggest  ...</p>
  ...
</div>

<div class="Footer">
  <p class="Disclaimer">These apocalyptic predictions  ...</p>
  <p>
    <a href="AboutUs.html">About Us</a>
    ...
  </p>
  <p>Copyright © 2011</p>
</div>
```

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Что означают эти многоточия?

Мы не можем поместить в эту книгу полностью всю разметку для каждого примера, по крайней мере не без того, чтобы она не разбухла до 12 тыс. страниц, для чего потребовалось бы вырубить целый лес. Но мы *можем* показать базовую структуру страницы и все ее основные элементы. Для этого во многих примерах мы пропускаем второстепенное содержимое, обозначая его посредством *многоточия* (...).

В качестве примера рассмотрим представленную выше разметку. Она содержит все тело показанной на рис. 2.2 страницы, но в ней отсутствует полный текст большинства абзацев, большая часть статьи, следующей после заголовка "Mayan Doomsday", и весь список ссылок в нижнем колонтитуле. Но, как уже упоминалось, весь пропущенный материал можно изучить в файлах для примеров, доступных на сайте **www.prosetech.com/html5**.

В хорошо написанной, традиционной HTML-странице (подобной этой) большинство работы по форматированию отдается на откуп таблице стилей посредством кон-

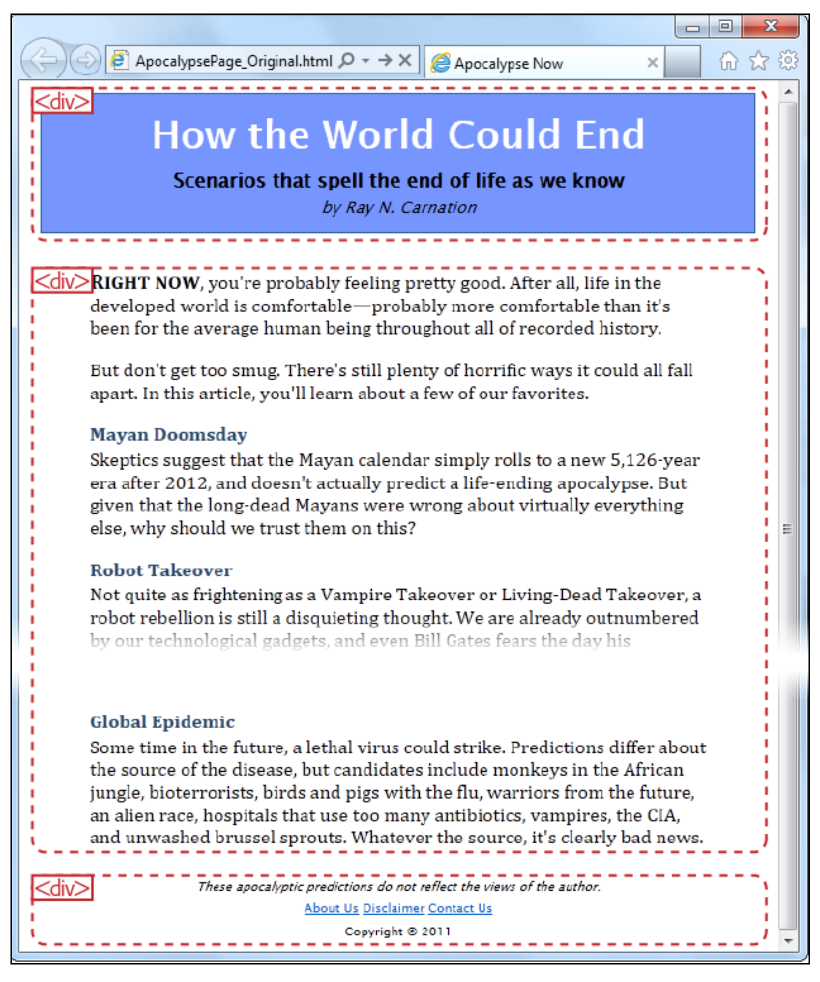


Рис. 2.2. Элементы `<div>` разбивают эту страницу на три логических блока: верхний колонтитул вверху, основное содержимое и нижний колонтитул внизу

тейнеров <div> и <span>. Элемент <span> позволяет форматировать отрывки текста внутри другого элемента. А элемент <div> — целые блоки содержимого, и устанавливает общую структуру страницы (рис. 2.2).

В данном случае перед таблицей стилей стоит легкая задача форматирования. Для всей страницы установлена максимальная ширина в 800 пикселов, чтобы текст не отображался длинными строками на широкоформатных мониторах. Верхний колонтитул помещен в блок синего цвета, для содержимого сделан отступ с каждой стороны, а нижний колонтитул расположен по центру страницы внизу.

Элемент <div> делает задачу форматирования легкой. Рассмотрим, например, таблицу стилей для форматирования блока верхнего колонтитула и его содержимого:

```
/* Форматируем элемент <div>, который представляет верхний колонтитул
   (как блок с рамкой и синей заливкой). */
.Header {
  background-color: #7695FE;
  border:      thin #336699 solid;
  padding:     10px;
  margin:      10px;
  text-align: center;
}

/* Форматируем все заголовки <h1> в элементе <div>
                верхнего колонтитула (заголовок статьи). */
.Header h1 {
  margin:    0px;
  color:     white;
  font-size: xx-large;
}

/* Форматируем подзаголовок в элементе <div> заголовка. */
.Header .Teaser {
  margin:      0px;
  font-weight: bold;
}

/* Форматируем строку с именем автора в элементе <div>. */
.Header .Byline {
  font-style: italic;
  font-size:  small;
  margin:     0px;
}
```

Обратите внимание на умелое использование в этом примере контекстуальных селекторов *(см. разд. "Контекстные селекторы" приложения 1)*. Например, с помощью селектора .Header h1 форматируются все элементы <h1> в блоке заголовка.

СОВЕТ

Этот пример также рассматривается в обзоре CSS в *приложении 1*. Если вы хотите подробно разобраться с правилами таблицы стилей, которые форматируют каждый блок, *см. разд. "Экскурсия по таблице стилей" приложения 1*.

Структурирование страницы с помощью HTML5

Элемент <div> — основное средство современного веб-дизайна. Это контейнер общего назначения, с помощью которого можно форматировать любую часть веб-страницы. Недостатком элемента <div> является то, что он не предоставляет никакой информации о странице. Встретив в разметке элемент <div>, вы (или браузер, средство разработки, скрин-ридер, поисковый робот и т. п.) понимаете, что нашли отдельный блок страницы, но не знаете назначение этого блока.

Чтобы исправить такую ситуацию, в HTML5 некоторые элементы <div> можно заменить более описательными семантическими элементами. Эти семантические элементы действуют в точности таким же образом, как и <div>-элемент: они группируют часть разметки в блок, но не выполняют никаких собственных операций над содержимым блока; они также предоставляют "стилевую зацепку", позволяющую присоединять форматирование. Но кроме всего этого, они также добавляют в страницу семантический смысл.

Далее приводится разметка страницы, показанной на рис. 2.1, но с применением новых элементов. В частности, два элемента <div> заменены HTML5-элементами <header> и <footer>.

```
<header class="Header">
  <h1>How the World Could End</h1>
  <p class="Teaser">Scenarios that spell the end of life as we know</p>
  <p class="Byline">by Ray N. Carnation</p>
</header>

<div class="Content">
  <p><span class="LeadIn">Right  now</span>, you're probably ...</p>
  <p>...</p>

  <h2>Mayan Doomsday</h2>
  <p>Skeptics  suggest  ...</p>
  ...
</div>

<footer class="Footer">
  <p class="Disclaimer">These apocalyptic predictions  ...</p>
  <p> <a href="AboutUs.html">About Us</a>
  ...
  </p>
  <p>Copyright © 2011</p>
</footer>
```

В этом примере вместо применяемых ранее элементов <div> используются элементы <header> и <footer>. Модифицирование большого веб-сайта можно начать с заключения существующих элементов <div> в соответствующие контексту элементы HTML5.

Обратите внимание, что элементы <header> и <footer> продолжают использоваться совместно с названиями классов. Таким образом, не нужно немедленно вносить исправления в таблицы стилей. Но названия классов кажутся избыточными, поэтому их лучше убрать:

```
<header>
  <h1>How the World Could End</h1>
  <p class="Teaser">Scenarios that spell the end of life as we know</p>
  <p class="Byline">by Ray N. Carnation</p>
</header>
```

Так как у страницы только один верхний колонтитул, его можно выделить именем элемента. В следующем листинге приведена модифицированная таблица стилей, которая форматирует все элементы <header>:

```
/* Форматируем <header> (как блок с рамкой и синей заливкой.) */
header {
  ...
}

/* Форматируем все заголовки <h1> в элементе <header>
   (заголовок статьи). */
header h1 {
  ...
}

/* Форматируем подзаголовок в элементе <header>. */
header .Teaser {
  ...
}

/* Форматируем строку с именем автора в элементе <header>. */
header .Byline {
  ...
}
```

Оба подхода одинаково действенные. Такая гибкость, без необходимости следовать жестким правилам, в HTML5 присуща многим решениям по разработке веб-страниц.

Обратите внимание на блок <div>, оставшийся в содержимом. Это вполне допустимо, т. к. HTML5-страницы часто содержат смесь семантических элементов и более общего контейнера <div>. Так как в HTML5 не существует отдельного элемента для содержимого, такого как, например, <content>, то значение обычного элемента <div> остается понятным.

> **Примечание**
>
> Данная веб-страница в ее настоящем виде не будет отображаться правильно в Internet Explorer более ранних версий, чем версия IE 9. Чтобы решить это проблему, необходимо применить простое обходное решение, рассматриваемое в *разд. "Браузерная совместимость для семантических элементов" далее в этой главе*. Но сначала ознакомьтесь еще с несколькими семантическими элементами, которые могут улучшить разметку страниц.

Наконец, есть еще один элемент, который стоит добавить в этот пример. Это элемент `<article>`, который представляет завершенную, самодостаточную единицу содержимого, например запись в блоге или новостную заметку. Элемент `<article>` обрамляет заголовок, строку с именем автора и основное содержимое. Добавив элемент `<article>` к разметке, получим следующую структуру:

```
<article>
  <header>
    <h1>How the World Could End</h1>
    ...
  </header>

  <div class="Content">
    <p><span class="LeadIn">Right now</span>, you're probably ...</p>
    <p>...</p>
    <h2>Mayan Doomsday</h2>
    <p>Skeptics suggest  ...</p>
    ...
  </div>
</article>

<footer>
  <p class="Disclaimer">These apocalyptic predictions  ...</p>
  ...
</footer>
```

Конечная структура страницы показана на рис. 2.3.

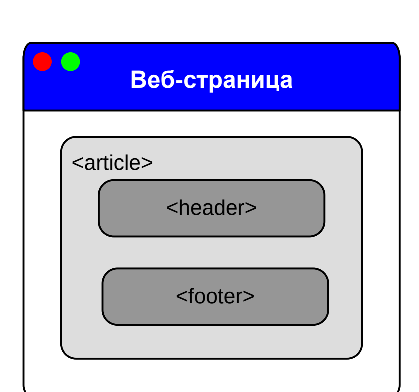


Рис. 2.3. Модифицированная разметка содержит три семантических элемента HTML5. Если логика старой структуры была такой: "Вот страница, разбитая на три раздела", то новая структура говорит: "Вот статья, в которой имеется нижний и верхний колонтитулы"

Хотя новая разметка отображается в браузере точно так же, как и старая, сама разметка содержит довольно много дополнительной информации. Например, заглянувший на ваш сайт поисковый робот по элементу <article> может быстро разобраться, где находится содержимое страницы, а по элементу <header> — где заголовок содержимого. Элемент <footer> не будет представлять для него интереса.

Примечание

Иногда статья может быть разбита на несколько веб-страниц. В настоящее время достигнуто соглашение, которое гласит, что каждая такая часть статьи с заголовком должна быть облачена в собственный элемент <article>, хотя она и не завершенная и не самодостаточная. Этот неуклюжий компромисс — всего лишь один из многих, на которые приходится идти, когда семантика встречается с практическими, презентационными аспектами Интернета.

Подзаголовки, созданные элементом *<hgroup>*

В предыдущем примере мы удачно применили элемент <header>. Но кроме этого элемента в HTML5 добавлен еще один элемент для работы с заголовками: <hgroup>. Официальные правила его применения следующие.

Прежде всего, для обычных отдельных верхних колонтитулов, не имеющих специального содержимого, вполне подойдет один из пронумерованных элементов заголовка — <h1>, <h2>, <h3> и т. д. Например:

```
<h1>How the World Could End</h1>
```

А заголовок и его подзаголовок можно вместе обернуть в элемент <hgroup>. (Но в таком случае не пытайтесь втиснуть туда что-либо другое, кроме пронумерованных элементов заголовка, т. е. <h1>, <h2>, <h3> и т. д.) Например:

```
<hgroup>
  <h1>How the World Could End</h1>
  <h2>Scenarios that spell the end of life as we know</h2>
</hgroup>
```

Увесистый верхний колонтитул, т. е., включающий кроме заголовка еще какое-то содержимое (например, краткое содержание статьи, дату публикации, имя автора, изображение или ссылки на подтемы), можно обернуть в элемент <header>:

```
<header>
  <h1>How the World Could End</h1>
  <p class="Byline">by Ray N. Carnation</p>
</header>
```

Наконец, если увесистый колонтитул также имеет подзаголовок, то чтобы обозначить дополнительное содержимое, элемент <hgroup> нужно вставить в элемент <header>:

```
<header>
  <hgroup>
    <h1>How the World Could  End</h1>
```

```
    <h2>Scenarios that spell the end of life as we know</h2>
  </hgroup>
  <p class="Byline">by Ray N. Carnation</p>
</header>
```

Это, конечно же, несколько модифицированное содержимое из предыдущего примера: вместо элемента <p> подзаголовок обозначен элементом <h2>.

С первого взгляда такая разметка может показаться потенциально сбивающей с толку. С точки зрения структуры она могла бы подразумевать, что все последующее содержимое является частью подраздела, который начинается с заголовка второго уровня, что не имеет особого смысла. В конце концов, кому нужна статья, состоящая целиком из одного большого подраздела? И даже если такая структура не влияет на отображение страницы в браузере, она меняет принцип создания *схемы документа* браузерами и другими инструментами *(см. разд. "Система HTML5 для создания схемы документа" далее в этой главе)*.

К счастью, эта проблема решается автоматически элементом <hgroup>. В структурном аспекте он обращает внимание *только* на заголовки верхнего уровня (в данном случае это <h1>). Другие заголовки отображаются в браузере, но они не включаются в схему документа. Такое поведение вполне логично, т. к. эти заголовки предназначены для обозначения подзаголовков, а не подразделов.

Вставка рисунков с помощью элемента *<figure>*

Многие веб-страницы оформляются изображениями. Но концепт *рисунка* несколько иной, чем изображения. Спецификация HTML5 советует рассматривать рисунки во многом подобно рисункам в книге, иными словами, изображение, отдельное от текста, но на которое в тексте делаются ссылки.

В общем, рисунки размещаются как *плавающие объекты*, т. е. их вставляют в первое удобное место в тексте, вместо того чтобы закреплять возле конкретного слова или элемента. Часто рисунки снабжены подписями, которые плавают вместе с ними.

В следующем листинге показан пример разметки HTML, которая добавляет рисунок к статье об апокалипсисе. (Код разметки также включает абзац непосредственно перед рисунком и абзац сразу же за ним, чтобы можно было видеть, как именно рисунок вставляется в разметку.)

```
<p><span class="LeadIn">Right now</span>, you're probably ...</p>

<div class="FloatFigure">
  <img src="human_skull.jpg" alt="Human skull">
  <p>Will you be the last person standing if one of these
     apocalyptic scenarios plays out?</p>
</div>

<p>But don't get too smug ...</p>
```

В разметке подразумевается, что для размещения рисунка используется правило в таблице стилей, которое также устанавливает поля, управляет форматированием подписи к рисунку и, необязательно, создает вокруг него рамку. В следующем листинге приведен пример такого правила:

```
/* Форматируем блок плавающего рисунка. */
.FloatFigure {
  float:          left;
  margin-left:    0px;
  margin-top:     0px;
  margin-right:   20px;
  margin-bottom:  0px;
}

/* Форматирует текст подписи к рисунку */
.FloatFigure p {
  max-width:      200px;
  font-size:      small;
  font-style:     italic;
  margin-bottom:  5px;
}
```

На рис. 2.4 показана работа этого кода.

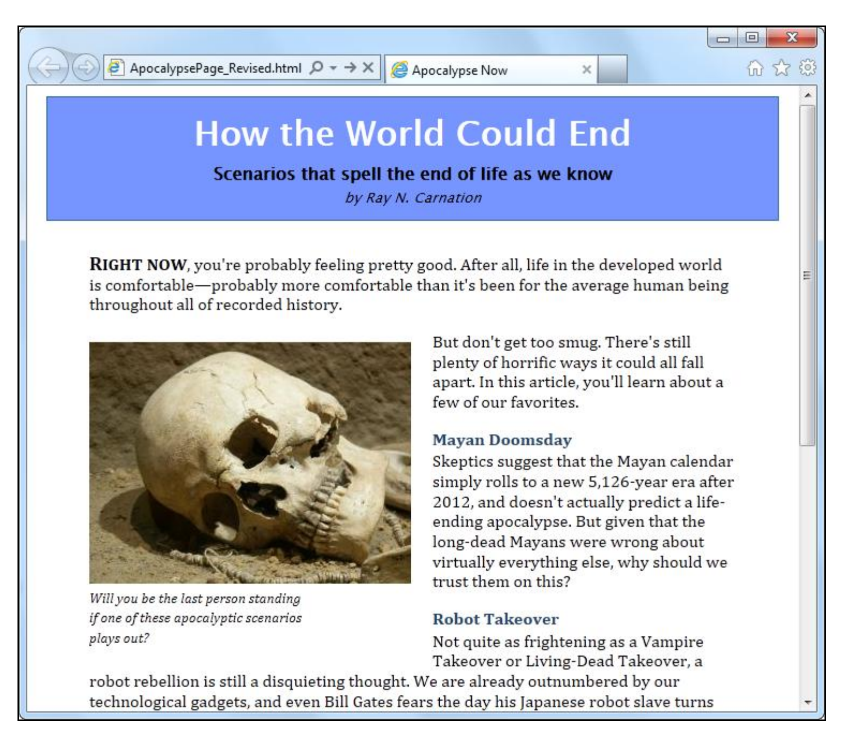


Рис. 2.4. Теперь статья украшена рисунком. В разметке рисунок определен сразу же за первым абзацем текста, поэтому он плавает слева от следующего текста. Обратите внимание, что ширина подписи к рисунку ограничена, чтобы создать красивый, плотный абзац

Если вам уже приходилось раньше создавать подобный код для вставки рисунка, то, возможно, будет интересно узнать, что HTML5 предоставляет новые семантические элементы, которые идеальным образом подходят для данного типа задач. В частности, вместо использования банального элемента <div> для контейнера рисунка можно использовать элемент <figure>. А текст подписи к рисунку помещается в элемент <figcaption> внутри элемента <figure> следующим образом:

```
<figure class="FloatFigure">
  <img src="human_skull.jpg" alt="Human  skull">
  <figcaption>Will you be the last person standing if one of
            these apocalyptic scenarios plays out?</figcaption>
</figure>
```

Конечно же, решение использовать ли таблицу стилей для размещения и форматирования блока рисунка остается за вами. (В данном примере это означает, что нужно изменить селектор выбора стиля, определяющий текст подписи к рисунку.) Сейчас в нем используется `.FloatFigure p`, но для модифицированного примера требуется `.FloatFigure figcaption`.

Совет

Обратите внимание, что элемент <figure> продолжает форматироваться на основе своего названия класса (FloatFigure), а не типа элемента. Это потому, что вы, скорее всего, захотите форматировать рисунки не одним, а несколькими способами. Например, может быть, вы захотите расположить рисунки слева, справа, с разными полями или параметрами подписи и т. п. Для такой гибкости рисунки следует форматировать с помощью классов.

В браузере рисунок продолжает отображаться точно таким же образом. Разница в разметке заключается в том, что теперь она совершенно понятна. (Кстати, возможности элемента <figcaption> не ограничиваются содержанием текста, в него можно помещать любые имеющие смысл элементы HTML, включая ссылки и пиктограммы.)

Наконец, стоит отметить, что в некоторых случаях подпись к рисунку может содержать полное его описание, что делает текст атрибута alt избыточным. В таком случае атрибут alt можно удалить из элемента <img>, как показано в следующем коде:

```
<figure class="FloatFigure">
  <img src="human_skull.jpg">
  <figcaption>A human skull lies on the sand</figcaption>
</figure>
```

Только ни в коем случае не удаляйте альтернативный текст, заменив его пустой строкой, т. к. это будет означать, что рисунок является чисто презентационным и программы чтения экрана должны игнорировать его.

Добавление боковой панели с помощью элемента *<aside>*

Новый элемент <aside> представляет содержимое, которое по смыслу связано с окружающим его текстом. Например, его можно использовать как врезку в печатной работе, чтобы ввести связанную тему или развить вопрос, исследуемый в основном документе. Логично применять элемент <aside> также в случаях, когда нужно где-то примостить блок объявлений, несколько ссылок на связанное содержимое или даже броскую цитату во врезке, подобно показанной на рис. 2.5.

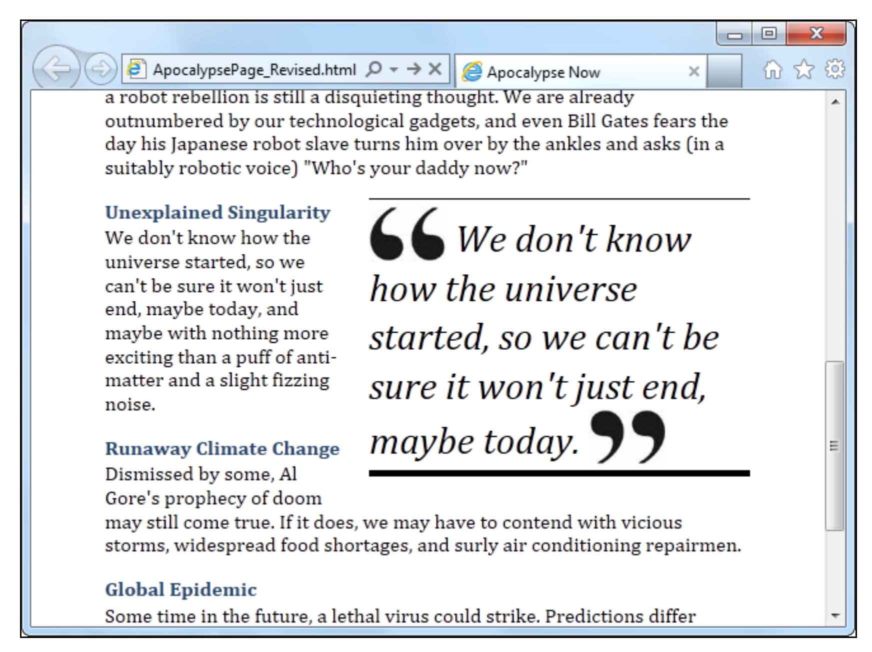


Рис. 2.5. Метод размещения во врезке броской цитаты позаимствован из книжной области. Такая цитата привлекает внимание читателя и выделяет важное содержимое

Этот эффект можно с легкостью создать с помощью хорошо приработанного элемента <div>, но элемент <aside> предоставляет более осмысленный способ делать разметку того же самого содержимого:

```
<p>...(in a suitably robotic voice) "Who's your daddy now?"</p>

<aside class="PullQuote">
  <img src="quotes_start.png" alt="Quote">
  We don't know how the universe started, so we can't be sure it
  won't just end, maybe today.
  <img src="quotes_end.png" alt="End quote">
</aside>

<h2>Unexplained Singularity</h2>
```

В этот раз таблица стилей врезает плавающую цитату справа:

```
.PullQuote {
  float:          right;
  max-width:      300px;
  border-top:     thin black solid;
  border-bottom:  thick black solid;
  font-size:      30px;
  line-height:    130%;
  font-style:     italic;
  padding-top:     5px;
  padding-bottom:  5px;
  margin-left:    15px;
  margin-bottom:  10px;
}
.PullQuote img {
  vertical-align: bottom;
}
```

Браузерная совместимость для семантических элементов

Пока все наши примеры использования новых возможностей HTML5 выглядят неплохо. Но результаты не будут такими воодушевляющими, если просмотреть нашу страницу в старых браузерах. Однако прежде чем заняться этим вопросом, стоит выяснить, какие браузеры предоставляют поддержку простых чисто семантических элементов. Эта информация предоставлена в табл. 2.1.

Таблица 2.1. *Поддержка браузерами семантических элементов*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
Минимальная версия	9	4	8	5	11.1	4	2.1

Как видим, ситуация в этой области не очень утешительная. К счастью, эта проблема из разряда легко решаемых. Ведь семантические элементы по сути ничего не делают. Все, что нужно браузеру делать для их поддержки, — это обрабатывать их как обычный элемент `<div>`. Но для этого нам нужно решить две проблемы.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Как создавались семантические элементы

Прежде чем приступить к созданию HTML5, его разработчики долго и тщательно изучали бесчисленное множество веб-страниц. Но они не просто просматривали популярные сайты, а изучили собранные Google статистические данные на свыше одного *миллиарда* веб-страниц. (Вы можете сами просмотреть эти данные на сайте **http://code.google.com/webstats**. Но для просмотра лучше пользоваться не Internet

Explorer, т. к. этот браузер не способен отобразить все стильные диаграммы и графики этого сайта.)

Компания Google проанализировала собранные ею данные и составила список названий классов, используемых веб-разработчиками в своих страницах. Идея состояла в том, что название класса может указать на назначение элемента и предоставить полезную информацию о том, каким образом разработчики структурируют свои страницы. Например, если у всех в разметке есть элемент `<div>`, в котором используется класс с названием *header*, тогда будет логичным допустить, что все разработчики помещают в свои веб-страницы верхние колонтитулы.

Первое, что исследователи компании Google узнали, было то, что в подавляющем большинстве страниц названия классов (или вообще таблицы стилей) не используются. Далее они составили короткий список наиболее часто употребляемых названий классов. Среди прочих, самыми популярными были такие названия классов, как *footer*, *header*, *menu*, *nav*, которые хорошо соответствовали новым семантическим элементам HTML5 `<footer>`, `<header>` и `<nav>`. Некоторые другие названия классов — такие как *search* и *copyright* — подсказывают возможные соответствующие семантические элементы.

Иными словами, Интернет изобилует одинаковыми базовыми дизайнами, например, страниц с верхним колонтитулом, нижним колонтитулом, боковыми панелями и меню навигации. Но при этом каждый разработчик смотрит немного по-своему на способ решения одной и той же задачи. От этого осознания всего лишь небольшой шаг к решению того, что в язык HTML можно добавить несколько новых элементов разметки, которая и так уже создается всеми разработчиками. Вот это озарение и вдохновило создание семантических элементов HTML5.

Первым делом нам нужно преодолеть привычку браузера рассматривать все неопознанные им элементы, как встроенные. Большинство семантических элементов HTML5 (включая все элементы, которые мы рассмотрели в этой главе, за исключением элемента `<time>`) являются блочными элементами. Это означает, что их нужно отображать в отдельной строке, с небольшим расстоянием между ними и предшествующими и следующими элементами.

Браузеры, которые не распознают элементы HTML5, не знают, что некоторые из этих элементов следует отображать как блочные, вследствие чего они, скорее всего, сбросят эти элементы в одну беспорядочную кучу. Чтобы решить это проблему, нужно просто добавить одно новое правило в таблицу стилей. Далее приведен код для суперправила, которое одним махом применяет форматирование блочного отображения к девяти требующим его элементам HTML5:

```
article, aside, figure, figcaption, footer, header, hgroup,
  nav, section, summary {
  display: block;
}
```

Это правило таблицы стилей никаким образом не будет влиять на браузеры, которые уже понимают HTML5, т. к. свойство `display` уже имеет значение `block`. Также оно не будет влиять ни на какие правила, которые уже используются для форматирования этих элементов, т. к. они будут продолжать применяться в дополнение к этому суперправилу.

Этот метод решает проблему распознавания семантических элементов в большинстве браузеров, но в это большинство не входят Internet Explorer 8 и более старые

его версии. Здесь нужно решить вторую проблему, заключающуюся в том, что эти версии IE отказываются применять форматирование таблицы стилей к элементам, которые они не распознают. К счастью, эта проблема решается с помощью небольшого трюка: браузер IE можно обмануть и заставить его распознавать чужие ему элементы, зарегистрировав их с помощью команды JavaScript. Например, в следующем JavaScript-коде браузер IE наделяется способностью понимать и применять стиль к элементу `<header>`:

```
<script>
  document.createElement('header')
</script>
```

Но вместо того чтобы самому разрабатывать такой код для всех элементов, можно воспользоваться уже готовым сценарием, ознакомиться с которым можно на сайте **http://tinyurl.com/nlcjxm**. Чтобы использовать этот сценарий, просто вставьте ссылку на него в блок `<head>` разметки:

```
<head>
  <title>...</title>
  <script
     src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
...
<head>
```

Этот код загружает сценарий с веб-сервера **html5shim.googlecode.com** и исполняет его, прежде чем браузер начинает обрабатывать остальную разметку страницы. Этот простой и краткий сценарий применит рассмотренный выше JavaScript-код для элемента `<header>`, с соответствующими изменениями, ко всем новым элементам HTML5, что позволяет форматировать их с помощью правил таблиц стилей. Добавьте в таблицу стилей приведенное ранее суперправило, и новые элементы будут отображаться должным образом, как блочные элементы. Вам только остается использовать новые элементы и добавить собственные правила таблицы стилей для их форматирования.

Кстати, сценарий html5.js исполняется, только если он определит, что используется старая версия Internet Explorer. Но если вы хотите вообще избежать накладных расходов, связанных с ненужным вызовом сценария JavaScript, можно в ссылку на него добавить соответствующее условие, как показано в следующем коде:

```
<!--[if lt IE 9]>
  <script
     src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
<![endif]-->
```

Таким образом, не нуждающиеся в нем браузеры (IE 9 и более поздние версии) будут игнорировать эту команду, что сэкономит несколько миллисекунд на исполнении кода разметки страницы.

Наконец, стоит отметить, что обычно Internet Explorer блокирует попытку исполнения файла страницы с диска локального компьютера. Это означает, что вместо ото-

бражения страницы IE выведет свою пресловутую панель безопасности, сообщая в ней о блокировке этого сценария. Чтобы исполнить сценарий, нужно щелкнуть по этой панели безопасности и в открывшемся диалоговом окне разрешить исполнение активного содержимого.

Хотя эта проблема не проявляется при исполнении файла страницы с веб-сервера, при локальном тестировании это, определенно, раздражает и отвлекает от главного. Решением будет добавить метку MOTW в начале страницы, как описано в конце *разд. "Добавление JavaScript-кода" главы 1*.

Совет

Альтернативным решением проблемы семантических стилей будет использование инструмента Modernizr *(см. разд. "Определение возможностей с помощью Modernizr" главы 1)*. Он автоматически решает эту проблему, не требуя применения сценария html5.js или правила таблицы стилей. Поэтому, если вы уже используете Modernizr для определения поддержки браузером определенных возможностей, можно считать проблему семантических стилей решенной.

Разработка сайта с использованием семантических элементов

Добавление семантических элементов в простую страницу, подобную документу, не составляет труда. Добавление их в завершенный веб-сайт ничуть не сложнее, но здесь возникает намного больше вопросов. А так как HTML5 является, по сути, неизведанной территорией, имеется лишь небольшое число соглашений (но большое число обоснованных расхождений во мнениях). Это означает, что при выборе между двумя возможными подходами к выполнению разметки, которые оба полностью отвечают требованиям стандарта HTML5, вам нужно самому решать, какое из них больше подходит для вашего содержимого.

В этом отношении, на рис. 2.6 показан более амбициозный пример страницы, создание которой мы рассмотрим далее.

В данном случае рассмотренная нами ранее одностраничная статья помещена в завершенный веб-сайт, структурированный по содержимому. Верх страницы содержит верхний колонтитул сайта, под которым размещено содержимое страницы. Левая боковая панель содержит элементы навигации, информацию о хозяине сайта (**About Us**) и рекламное изображение.

Верхние колонтитулы

Элемент `<header>` используется в двух похожих ситуациях. Первая, когда дается верхний колонтитул какому-либо содержимому, а вторая, когда дается верхний колонтитул веб-странице. Иногда эти две задачи перекрываются, как в примере с отдельной статьей, показанной на рис. 2.1. Но в других случаях приходится иметь дело с веб-страницей, которая имеет как верхний колонтитул страницы, так и один или несколько блоков содержимого со своими верхними колонтитулами. Пример такой страницы и показан на рис. 2.6.



Рис. 2.6. Страница HTML5 с более сложной структурой

Эта ситуация несколько посложнее тем, что соглашения по использованию элемента <header> зависят от его роли. Для содержимого мы, скорее всего, верхний колонтитул применять не будем, если только он не требуется по какой-либо причине. А единственной причиной, по которой он может потребоваться, будет создание так называемого "увесистого" верхнего колонтитула, т. е. колонтитула, содержащего кроме заголовка дополнительную информацию. Если же требуется обычный заголовок, лучше всего использовать отдельный элемент <h1>. Если требуется только заголовок с одним или несколькими подзаголовками, можно использовать сам элемент <hgroup>. Но если нужно добавить прочие подробности, имеет смысл облачить весь "пакет" в элемент <header>, как рассматривается в *разд. "Структурирование страницы с помощью HTML5" ранее в этой главе*. Но, создавая верхний колонтитул для веб-сайта, многие разработчики оборачивают его в элемент <header>, даже если он не содержит ничего, кроме заголовка в рамке с CSS-форматированием. В конце концов, это один из основных аспектов дизайна веб-сайта, и кто его знает, может быть, когда-либо придется добавить в него что-то новое.

Вывод следующий: страницы могут иметь несколько элементов <header> (и часто будут иметь их), даже если они играют разные роли на странице.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Превращение веб-страницы в веб-сайт

На рис. 2.6 показана отдельная страница гипотетического веб-сайта.

Страницы настоящего веб-сайта, которых может быть от нескольких единиц до нескольких десятков, будут иметь такую же организацию и такую же боковую панель. Единственная разница между всеми этими страницами будет в их основном содержимом, которым в данной странице является статья.

В HTML5 нет какой-либо волшебной палочки для превращения веб-страницы в веб-сайт. Поэтому приходится использовать те же приемы и методы, что и в традиционном HTML.

- **Серверные инфраструктуры.** Эта простая идея заключается в следующем. Когда браузер запрашивает страницу, веб-сервер собирает ее из отдельных частей, включая общие элементы (такие как меню навигации) и конкретное содержимое. Это наиболее распространенный подход и единственный, который следует использовать для больших, профессиональных веб-сайтов. Этот подход реализуется массой разных способов разнообразных технологий, от устаревших расширений на серверной стороне до платформ развитых веб-приложений (таких как ASP.NET и PHP) и систем управления информационным наполнением (таких как Drupal и WordPress).

- **Шаблоны страниц.** Мощные инструменты для создания и редактирования веб-страниц, такие как Dreamweaver и Expression Web, содержат средство для создания шаблонов страниц. Сначала создается шаблон, в котором определяется структура веб-страниц, и который содержит информацию, необходимую для вывода на каждой странице, например колонтитулы и боковую панель. Потом на основе этого шаблона создаются все страницы сайта. Что особенно полезно, т. к. при обновлении шаблона веб-редактор автоматически обновляет все страницы на его основе.

Конечно же, вы можете использовать любой из этих подходов, поэтому в данной книге основное внимание уделяется конечному результату: собранной вместе разметке, которая составляет завершенную страницу, выводимую в окне браузера.

По сравнению с простой страницей на рис. 2.6 добавлен верхний колонтитул веб-сайта. Этот колонтитул состоит из отдельного изображения-баннера, содержащего графический текст и изображение. Код соответствующего элемента `<header>` выглядит следующим образом:

```
<header class="SiteHeader">
  <img src="site_logo.png" alt="Apocalypse Today">
  <h1 style="display:none">Apocalypse Today</h1>
</header>
```

Сразу же можно заметить, что этот колонтитул добавляет элемент, который не отображается на странице: заголовок, который дублирует текст в изображении. Но настройка в строке кода скрывает этот заголовок.

Это поднимает очевидный вопрос: какой смысл в добавлении в разметку заголовка, который не отображается в браузере? Как ни странно, этому есть несколько причин. Во-первых, требуется, чтобы элемент `<header>` содержал заголовок какого-либо уровня, просто чтобы отвечать требованиям HTML5. Во-вторых, такой дизайн делает страницу более доступной для людей, которые перемещаются по ней с помощью программы чтения экрана, т. к. они часто перепрыгивают с одного заголов-

ка на другой, не обращая внимания на содержимое между ними. И в-третьих, таким образом определяется структура заголовков, которую можно использовать в оставшейся части страницы. Попросту говоря, если вы начнете с элемента `<h1>` для колонтитула веб-сайта, то для заголовков других разделов страницы (например "Articles" и "About Us" в боковой панели) можно использовать элементы `<h2>`. Более подробно этот аспект дизайна рассматривается *во врезке "Часто задаваемый вопрос. Структура заголовков сайта" далее в этой главе*.

ПРИМЕЧАНИЕ

Конечно же, можно упростить себе жизнь, создав обычный текстовый верхний колонтитул. (И если вам хочется вычурных украшений, можно воспользоваться новой возможностью CSS3 для встроенных шрифтов, которая рассматривается в *разд. "Типография для Интернета" главы 8*.) Но для многих веб-страниц, которые используют заголовок в виде изображения, применение метода скрытия заголовка будет следующим лучшим решением.

Создание навигационных ссылок с помощью элемента *<nav>*

Самой интересной особенностью нашего гипотетического веб-сайта является левая боковая панель, которая содержит средства навигации по сайту, дополнительную информацию и рекламное изображение. (Обычно для размещения рекламного изображения применяется сценарий JavaScript, который произвольно выбирает рекламу с помощью специальных служб, таких как, например, Google AdSense. Но в этом примере место для рекламы просто жестко зарезервировано с помощью конкретного изображения.)

В веб-сайте, созданном на традиционном HTML, вся боковая панель была бы помещена в элемент `<div>`. Но в HTML5 для этого почти всегда используются два более специфичных элемента: `<aside>` и `<nav>`.

Элемент `<aside>` несколько схож с элементом `<header>` в том отношении, что он имеет тонкое, слегка растяжимое значение. Его можно использовать для обозначения единицы несвязанного содержимого, как в случае с броской цитатой во врезке в *разд. "Добавление боковой панели с помощью элемента <aside>" ранее в этой главе* (см. рис. 2.5). Также с его помощью можно обозначать совершенно отдельный блок, т. е. стоящий в стороне от основной структуры страницы.

ЧАСТО ЗАДАВАЕМЫЙ ВОПРОС
Структура заголовков сайта

Допустимо ли иметь на странице несколько заголовков первого уровня? Хорошая ли это идея?

Согласно официальным правилам HTML, можно иметь сколько угодно заголовков первого уровня. Но веб-разработчики часто стремятся иметь только один заголовок первого уровня на странице, т. к. это улучшает характеристики доступности сайта. (Причиной этому является то обстоятельство, что пользователи, просматривающие сайт с помощью средств чтения экрана, при переходах с одного заголовка второго уровня на другой могут пропустить какой-либо заголовок первого уровня.) Также су-

ществует некая позиция веб-мастеров, согласно которой каждая страница должна иметь ровно один заголовок, который является однозначным по всему веб-сайту и ясно указывает поисковым движкам, какое содержимое их ожидает.

В примере на рис. 2.6 используется этот стиль. Заголовок "Apocalypse Today" вверху сайта является единственным заголовком `<h1>` на этой странице. В других блоках страницы, таких как "Articles" и "About Us" в боковой панели, используются заголовки второго уровня. Название статьи также оформлено заголовком второго уровня. (Осуществив дополнительное планирование, можно варьировать текст заголовка первого уровня в соответствии с содержимым текущей статьи. В конце концов, этот заголовок не отображается, но может улучшить шансы попадания страницы в результаты более уточненного запроса поискового движка, такого как Google.)

Существуют и другие, такие же действенные подходы. Например, можно использовать заголовки первого уровня для названий каждого основного блока страницы, включая боковую панель, статью и т. п. Альтернативно заголовок веб-сайта можно сделать первого уровня, а заголовки боковой панели — второго уровня, но заглавие статьи сделать вторым заголовком первого уровня. Все это прекрасно работает в HTML5 благодаря его новой системе создания схемы страницы. Как мы увидим в *разд. "Блоки" далее в этой главе*, некоторые элементы, включая элемент `<article>`, рассматриваются как отдельные блоки, которые имеют собственные индивидуальные иерархические структуры. Поэтому будет совершенно логичным, чтобы в этих блоках иерархия заголовков начиналась сначала с помощью нового элемента `<h1>`. (Но стандарт HTML5 также разрешает начинать новую иерархию заголовков с другого их уровня.)

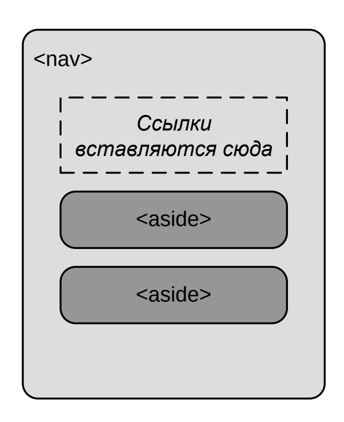
Короче говоря, нет одного определенного ответа, какой должна быть структура вашего веб-сайта. Выглядит вероятным, что подход с использованием нескольких заголовков первого уровня будет все более популярным по мере завоевания Интернета стандартом HTML5. Но пока многие веб-разработчики продолжают применять подход с использованием одного заголовка первого уровня, чтобы не усложнять жизнь программам чтения экрана.

Элемент `<nav>` служит оболочкой для блока ссылок, указывающих на тематические разделы текущей страницы или на другие страницы веб-сайта. Большинство страниц имеет несколько блоков `<nav>`. Но не все ссылки требуется помещать в блок `<nav>`; обычно этот блок резервируется для самых больших и наиболее важных разделов навигации страницы. Например, для списка статей (как на рис. 2.6) определенно требуется блок `<nav>`. Но, скажем, для пары ссылок внизу страницы на условия лицензирования и контактную информацию полноценный блок `<nav>` не является необходимым.

Теперь подходящее время попробовать попрактиковаться в использовании этих двух элементов. Сначала ознакомьтесь внимательно с содержимым боковой панели на рис. 2.6. Потом набросайте на бумаге разметку структуры этого содержимого. Далее продолжайте ее разработку с чтением этой главы, чтобы определить наилучшее возможное решение.

Содержимое этой боковой панели можно структурировать, по крайней мере, двумя способами, которые показаны на рис. 2.7.

Слева на рис. 2.7 показано, что всю боковую панель можно рассматривать как панель навигации, в которую также втиснуто немного другого содержимого. В этом случае вся панель может быть взята в элемент `<nav>`, а для других блоков ее содержимого использованы элементы `<aside>` (т. к. эти блоки не связаны с основным содержимым боковой панели, которым являются ссылки).



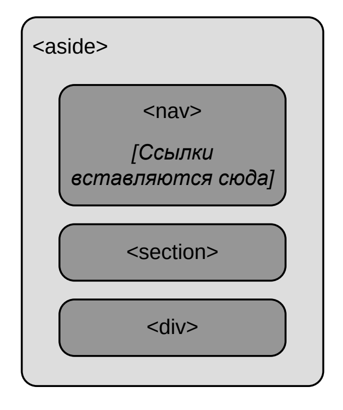


Рис. 2.7. Два способа структурирования содержимого боковой панели

Альтернативно, как показано на рис. 2.7 справа, всю боковую панель можно рассматривать как отдельный блок веб-страницы, который имеет несколько назначений. В этом случае вся боковая панель заключается в элемент `<aside>`, а навигационные ссылки внутри этого элемента берутся в элемент `<nav>`.

Боковая панель нашего гипотетического сайта на рис. 2.6 построена с применением второго подхода. Этот подход был избран потому, что в данном случае боковая панель служит нескольким целям, ни одна из которых не является определенно доминирующей. Но для боковой панели с большим и сложным блоком навигации (наподобие многоуровневого разворачивающегося меню) и небольшим второстепенным содержимым первый подход может быть более логичным.

В следующем листинге приведен код для структурирования боковой панели, разбивающий ее на три блока:

```
<aside class = "NavSidebar">
  <nav>
    <h2>Articles</h2>
    <ul>
      <li><a href="...">How The World Could End</a></li>
      <li><a href="...">Would Aliens Enslave or Eradicate Us?</a></li>
      ...
    </ul>
  </nav>

  <section>
    <h2>About Us</h2>
    <p>Apocalypse Today is a world leader in conspiracy theories ...
    </p>
  </section>

  <div>
    <img src="ad.jpg" alt="Luckies cigarette ad: it's toasted">
  </div>
</aside>
```

Обратите внимание на следующие ключевые моменты.

- **Блоки заголовков ("Articles" и "About Us") являются заголовками второго уровня.** Таким образом, они явно на более низком уровне иерархии, чем заголовок веб-сайта первого уровня, что улучшает доступность страницы для программ чтения экрана.
- **Ссылки выполнены в виде маркированного списка с помощью элементов <ul> и <li>.** Веб-разработчики сходятся во взглядах, что список является наилучшим средством для создания наиболее доступного навигационного меню из набора ссылок. Но может потребоваться применить правила таблицы стилей, чтобы удалить отступы (как было сделано в данном случае) и маркеры списка (что в этом примере не было сделано).
- **Блок "About Us" вставлен в элемент <section>.** Этот элемент был использован в связи с отсутствием какого-либо другого семантического элемента, более подходящего для данного содержимого. Элемент <section> более специфичен, чем элемент <div>; он подходит для обозначения любого блока содержимого, которое начинается с заголовка. Если бы существовал более определенный элемент для данного контекста, например гипотетический элемент <about>, он бы подошел лучше, чем базовый элемент <section>, но такого элемента не существует.
- **Рекламное изображение заключено в элемент <div>.** Элемент <section> годится только для содержимого, начинающегося с заголовка, а данный блок изображения заголовка не имеет. (Хотя если бы заголовок имелся, например "Слово нашим спонсорам", то элемент <section> был бы лучшим выбором.) С технической точки зрения нет надобности вставлять изображение в какой-либо элемент, но использование элемента <div> облегчает задачу отделения и форматирования этого блока, а также, если необходимо, вставку кода JavaScript для манипулирования им.

Боковую панель можно облагородить дополнительными подробностями, которых нет в данном примере. Например, сложные боковые панели всегда начинаются с элемента <header> и заканчиваются элементом <footer>. Также они могут содержать несколько блоков <nav>, например, один для списка заархивированного содержимого, другой для списка последних новостных сообщений, третий для списка блогов или подобных сайтов и т. д. Одним из примеров такой панели может быть боковая панель типичного блога, битком забитая блоками, многие из которых содержат ссылки.

Для форматирования боковой панели, созданной с помощью элемента <aside>, применяются те же правила таблиц стилей, что и для традиционной боковой панели, созданной с помощью элемента <div>. Эти правила указывают место размещения боковой панели методом абсолютного позиционирования, а также другие аспекты форматирования, такие как поля панели и цвет фона. Далее приведен пример такого правила таблицы стилей:

```
aside.NavSidebar
{
  position:         absolute;
  top:              179px;
```

```
  left:             0px;
  padding:          5px 15px 0px 15px;
  width:            203px;
  min-height:       1500px;
  background-color: #eee;
  font-size:        small;
}
```

Кроме этого правила применяются другие, контекстуальные, для форматирования элементов `<h2>`, `<ul>`, `<li>` и `<img>` боковой панели. (Как и для всех других примеров в этой книге, на странице **www.prosetech.com/html5** можно загрузить для ознакомления код этого примера, включающий полную таблицу стилей.)

Теперь, когда вы понимаете, каким образом создается боковая панель, можете посмотреть, как она интегрируется в структуру всей страницы (рис. 2.8).

ПРИМЕЧАНИЕ

Как мы узнали, элемент `<nav>` часто используется самостоятельно, а также может заключаться в элемент `<aside>`. Есть еще одно место, где он часто встречается: в элементе `<header>` колонтитула веб-страницы.

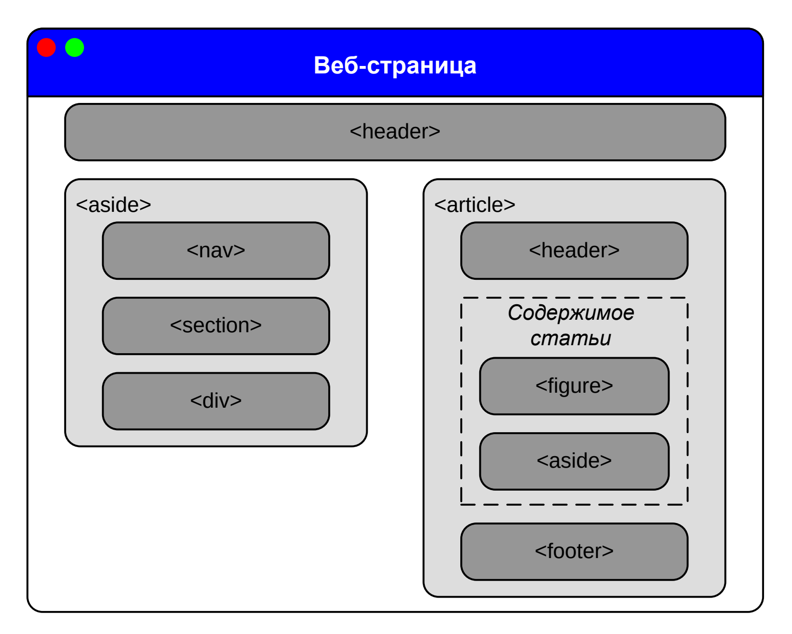


Рис. 2.8. Все семантические элементы сайта апокалипсиса (см. рис. 2.6)

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ

Создание разворачиваемых блоков с помощью элементов <details> и <summary>

Несомненно, вы часто видели на веб-сайтах разворачиваемые блоки: разделы содержимого, которые можно показать или скрыть, щелкнув на заголовке. Создание разворачиваемого блока является одним из самых легких трюков веб-дизайна, который

можно провернуть с помощью базового сценария JavaScript. Нужно просто отреагировать на щелчок по заголовку и изменить параметры стиля, чтобы скрыть блок. Код для этого может быть таким:

```
var box = document.getElementById("myBox");
box.style.display = "none";
```

Повторный щелчок отображает блок. Код для этого почти такой же, как для скрытия:

```
var box = document.getElementById("myBox");
box.style.display = "block";
```

В этом отношении представляет интерес то обстоятельство, что HTML5 добавляет два семантических элемента, предназначенных для автоматизации этого поведения. Идея заключается в том, что разворачиваемый блок вставляется в элемент <details>, а заголовок блока — в элемент <summary>. Получим разметку наподобие следующей:

```
<details>
  <summary>Блок № 1</summary>
  <p>Если вы видите этот текст, блок развернут</p>
</details>
```

Браузеры, которые поддерживают эти элементы (в момент написания этой книги такую поддержку предоставляет только браузер Chrome), показывают только заголовок, возможно, украшенный какой-либо безделицей типа небольшого треугольничка рядом с заголовком. Когда пользователь щелкает по заголовку, блок разворачивается, показывая все свое содержимое. Браузеры без поддержки элементов <details> и <summary> покажут все содержимое с самого начала, не предоставляя пользователю никакой возможности свернуть его.

Отношение веб-разработчиков к элементам <details> и <summary> несколько противоречивое. Многие из них считают, что эти элементы не совсем семантические, т. к. они имеют дело больше со стилем, чем с логической структурой.

Но пока лучше всего избегать использования элементов <details> и <summary> из-за такого низкого уровня их поддержки браузерами. Хотя можно было бы написать обходное JavaScript-решение для браузеров, не поддерживающих этих элементов, но для этого потребуется больше усилий, чем для написания нескольких строк JavaScript для выполнения операций сворачивания/разворачивания самостоятельно и к тому же на любом браузере.

Нижние колонтитулы

Язык HTML5 и увесистые верхние колонтитулы были предназначены друг для друга. В верхние колонтитулы можно не только втиснуть подзаголовки и имя автора, но также вставлять изображения, навигационные блоки (с помощью элемента <nav>) и практически все, что угодно другое, которому место вверху страницы.

Как ни странно, HTML5 не так услужлив в отношении нижних колонтитулов. По его мнению, содержание нижних колонтитулов должно быть ограничено несколькими основными подробностями, такими как авторские права, авторство, правовые ограничения и отдельные ссылки. Нижние колонтитулы не должны содержать длинных списков ссылок, блоков важного содержимого или несвязанных с содер-

жимым сайта элементов, таких как реклама, кнопок социальных сетей или элементов интерфейса веб-сайта.

В связи с этим возникает следующий вопрос: что делать, если дизайн веб-сайта требует увесистого нижнего колонтитула? В конце концов, в настоящее время увесистые нижние колонтитулы пользуются большой популярностью в области веб-дизайна (см. рис. 2.9 для примера).



Рис. 2.9. Этот раздутый до абсурда нижний колонтитул напичкан крикливыми побрякушками, наподобие изображения о присуждении награды и кнопок социальных сетей. В нем применяется фиксированное расположение, закрепляющее его внизу окна браузера, наподобие панели инструментов. Этот колонтитул оправдывает лишь одно положительное свойство: кнопка закрытия в правом верхнем углу, с помощью которой от него можно избавиться в любое время

В увесистых нижних колонтитулах используется несколько замысловатых технологий, включая перечисленные далее.

- **Фиксированное позиционирование**, чтобы они всегда были закреплены у нижнего края окна браузера, несмотря на вертикальную прокрутку страницы. На рис. 2.9 показан пример фиксированного нижнего колонтитула.
- **Кнопка закрытия**, чтобы пользователь мог закрыть его после прочтения и увеличить область просмотра (см. рис. 2.9). Эта операция реализуется с помощью простого сценария JavaScript, скрывающего элемент, в который вставлен нижний колонтитул. В качестве примера такого кода может служить код из предыдущего раздела.
- **Полупрозрачный фон**, чтобы можно было видеть содержимое страницы *сквозь* нижний колонтитул. Это дает хороший эффект, если в нижнем колонтитуле раз-

мещена реклама или важное предупреждение, и обычно используется совместно с кнопкой закрытия.

- **Анимация**, чтобы колонтитул всплывал или выскальзывал при определенных условиях. См., например, диалоговое окно со ссылкой на связанную статью, которое выскальзывает с правой стороны экрана при достижении конца статьи на www.nytimes.com.

Если дизайн вашего веб-сайта требует такого нижнего колонтитула, у вас есть выбор. Самый простой подход — игнорировать правила. Он не так ужасен, как может показаться, т. к. другие веб-разработчики наверняка создают такие же раздутые нижние колонтитулы, и со временем официальные правила могут быть ослаблены, разрешая эти колонтитулы. Но если в настоящее время вы хотите придерживаться требований стандарта, вам нужно немного приспособить свою разметку. К счастью, это не слишком трудная задача.

Трюк заключается в том, чтобы отделить стандартные элементы нижнего колонтитула от дополнительных. В браузере эти элементы будут отображаться как цельный нижний колонтитул, но в разметке они не будут относиться к элементу <footer>. Например, в следующем листинге показан код для реализации этим методом нижнего колонтитула на рис. 2.9:

```
<div id="FatFooter">
  <!-- Сюда вставляется дополнительное содержимое нижнего колонтитула. -->
  <img onclick="CloseBox()" src="close_icon.png" class="CloseButton">
  ...
  <footer>
    <!-- Сюда вставляется стандартное содержимое нижнего колонтитула. -->
    <p>The views expressed on this site do not ... </p>
  </footer>
</div>
```

Внешний элемент <div> не имеет семантического значения. Он просто используется, как удобный контейнер для связки дополнительного, "увесистого" содержимого со стандартным содержимым нижнего колонтитула. Он также позволяет применить следующее правило форматирования из таблицы стилей, фиксирующее увесистый нижний колонтитул в одном месте:

```
#FatFooter {
  position: fixed;
  bottom:   0px;
  height:   145px;
  width:    100%;
  background: #ECD672;
  border-top: thin solid black;
  font-size:  small;
}
```

Примечание

В этом примере правило таблицы стилей накладывает указанное в нем форматирование по идентификатору (используя селектор #FatFooter), а не по названию класса

(используя, например, селектор .FatFooter). Это потому, что для увесистого нижнего колонтитула уже требуется однозначный идентификатор, чтобы сценарий JavaScript мог найти и скрыть его при нажатии кнопки закрытия. Логичнее использовать этот однозначный идентификатор в таблице стилей, чем добавлять для этой же цели название класса.

Как вариант, нижний колонтитул можно также поместить в элемент <aside> с тем, чтобы ясно указать, что его содержимое представляет отдельный блок, связанный по смыслу с остальным содержимым страницы. Такая структура будет выглядеть следующим образом:

```
<div id="FatFooter">
  <aside>
    <!-- Сюда вставляется дополнительное содержимое
                                          нижнего колонтитула. -->
    <img onclick="CloseBox()" src="close_icon.png" class="CloseButton">
    ...
  </aside>
  <footer>
    <!-- Сюда вставляется стандартное содержимое нижнего колонтитула. -->
    <p>The views expressed on this site do not... </p>
  </footer>
</div>
```

Важным обстоятельством здесь является то, что элемент <footer> не вставляется в элемент <aside> по той причине, что элемент <footer> относится не к элементу <aside>, а ко всему веб-сайту. Если какой-либо элемент <footer> относится к некоторой единице содержимого, его нужно поместить внутри элемента, в который обернуто данное содержимое.

Примечание

Правила и руководства по надлежащему использованию семантических элементов HTML5 продолжают развиваться. Вопросы о правильном способе разметки больших, сложных сайтов порождают горячие споры в HTML-сообществе. Наилучшим советом будет следующий: если вы не уверены, что какой-либо метод является правильным, не применяйте его. Можно также обсудить этот вопрос на каком-либо сайте, где вы сможете обменяться мнением по нему с десятком суперумных HTML-гуру. Одним, особенно хорошим, сайтом является сайт **http://html5doctor.com**, на котором можно ознакомиться с продолжающимися дебатами в комментариях к большинству статей.

Блоки

Как мы уже узнали, к использованию семантического элемента <section> прибегают в самом крайнем случае. Например, блок содержимого с заголовком, для которого не подходят другие семантические элементы, обычно лучше поместить в элемент <section>, чем в элемент <div>.

Но какое содержимое помещается в типичный блок <section>? В зависимости он конкретной точки зрения, элемент <section> можно считать либо гибким средством для искусного решения многих разнообразных задач, либо кувалдой, которая тоже

решает многие задачи, но уже не так искусно. Причиной этому то обстоятельство, что в веб-странице элемент <section> играет разнообразные роли. Он может помечать любой из следующих видов содержимого:

- небольшие блоки содержимого, отображаемые вместе с главной страницей, наподобие блока "About Us" в веб-сайте апокалипсиса;
- автономные блоки содержимого, которые нельзя, по сути, назвать статьей, наподобие счетов клиентам или списка продуктов;
- группы содержимого, например набор статей на сайте новостей;
- *часть* документа. Например, в статье об апокалипсисе каждый из сценариев конца света можно выделить отдельным блоком. Иногда элемент <section> используется таким образом для того, чтобы обеспечить правильную иерархическую схему документа *(см. следующий раздел)*.

Два последних пункта списка вызывают удивление. Многие веб-разработчики считают использование одного элемента для работы как с небольшой частью статьи, так и с целой группой статей недопустимым. Некоторые полагают, что для работы с этими двумя разными сценариями HTML5 должен иметь, по крайней мере, два разных элемента. Но создатели HTML5 решили для простоты ограничить количество новых элементов, в то же самое время, делая новые элементы насколько возможно гибкими.

И последнее, что следует иметь в виду. Элемент <section> также влияет на иерархическую схему веб-страницы, понятие которой рассматривается в следующем разделе.

Система HTML5 для создания схемы документа

В HTML5 определен набор правил, которые устанавливают требования для создания *схемы структуры документа* для любой веб-страницы. Схема структуры документа может быть полезна в разных ситуациях. Например, браузер может перескакивать с одной части схемы на другую. Или с помощью инструмента для разработки веб-страниц можно расположить блоки в новом порядке, перетаскивая и оставляя их в нужном месте в окне схемы страницы. Либо поисковый движок может использовать схему страницы, чтобы создать более удобное ее представление для предпросмотра. Но наиболее полезной схема будет для средств чтения экрана, которые могут использовать ее, чтобы проводить пользователей с ограниченным зрением по иерархии разделов и подразделов, имеющей большую глубину вложения.

Но ни один из этих сценариев не был еще реализован на практике, т. к. в настоящее время почти никто не использует схему страницы. Исключение — небольшой набор инструментов разработки, которые рассматриваются в следующем разделе.

ПРИМЕЧАНИЕ

Трудно восхищаться возможностью, которая не влияет на отображение страницы браузером и не используется другими инструментами. Тем не менее полезно проверять схему разрабатываемых веб-страниц (или, по крайней мере, схему типичной страницы сайта), чтобы быть уверенным в логичности структуры страниц и соблюдении правил HTML5.

Как просмотреть схему веб-страницы?

Чтобы действительно понимать схемы страниц, нужно ознакомиться с ними в собственной разработке. В настоящее время ни один из браузеров не поддерживает правила схем HTML5 и не оснащен средством для просмотра таковых. Но этот пробел можно заполнить несколькими специальными инструментами.

- **Онлайновый построитель HTML-схем.** Откройте страницу **http://gsnedders.html5.org/outliner** и предоставьте построителю схем страницу для обработки. Как и в случае с валидатором HTML5 *(см. разд. "Проверка кода HTML5" главы 1)*, страницу можно предоставить тремя способами: загрузив файл с диска своего компьютера, указав URL страницы или вставив разметку страницы в текстовое поле. После предоставления файла любым из этих способов для создания схемы нужно нажать кнопку **Outline this!** соответствующего способа.
- **Расширение браузера Chrome.** В браузере Chrome схему страницы можно просмотреть с помощью подключаемого модуля h5o. Загрузите этот модуль с сайта **http://code.google.com/p/h5o** и установите его в браузере. После установки можно будет просматривать схемы страниц, предоставляемых веб-сервером, но, к сожалению, на момент написания этих строк, не файлов, открываемых с диска локального компьютера. При посещении страницы справа от строки адреса выводится значок построителя схем, щелчок по которому выводит схему про

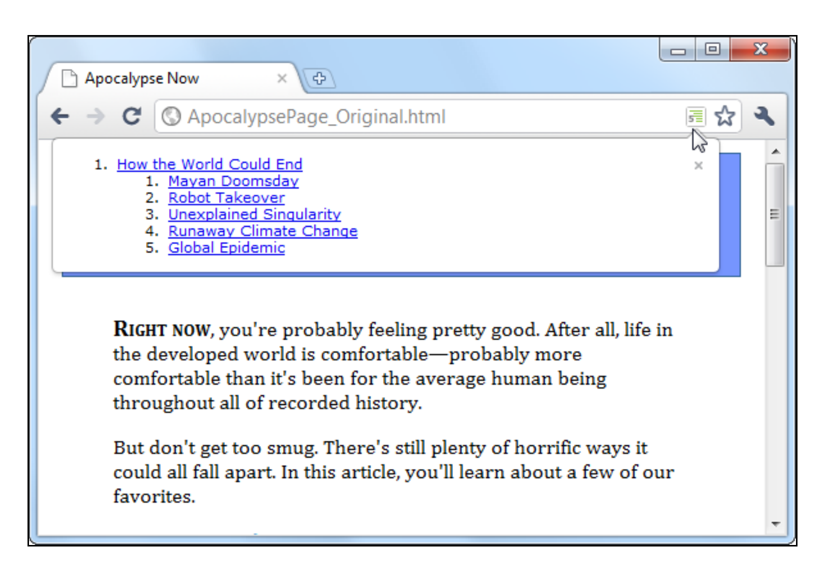


Рис. 2.10. При посещении веб-страницы справа от адресной строки браузера выводится значок построителя схем, при щелчке по которому выводится окно со схемой этой страницы

сматриваемой страницы (рис. 2.10). На сайте h5o можно также загрузить JavaScript-сценарий *bookmarklet* для браузеров Firefox и Internet Explorer. Этот сценарий добавляется в список закладок браузера, после чего в нем можно просматривать схему просматриваемой страницы.

- **Расширения для браузера Opera.** Существует также версия построителя схем h5o для браузера Opera. Загрузить ее можно с того же самого сайта, что и версию для Chrome, или установить непосредственно с сайта **http://tinyurl.com/3k3ecdy**.

Базовые схемы

Чтобы создать мысленную картину схемы веб-страницы, представьте ее лишенной всего содержимого, за исключением текста в элементах пронумерованных заголовков (<h1>, <h2>, <h3> и т. д.). Потом сместите эти заголовки вправо на расстояние, зависящее от уровня каждого заголовка. Таким образом, отступ более глубоко вложенных заголовков будет большим, чем заголовков более высокого уровня.

Рассмотрим для примера статью об апокалипсисе в ее первоначальном состоянии, до того как она была подкорректирована в соответствии со стандартом HTML5:

```
<body>
  <div class="Header">
    <h1>How the World Could End</h1>
    ...
  </div>
  ...
  <h2>Mayan Doomsday</h2>
  ...
  <h2>Robot Takeover</h2>
  ...
  <h2>Unexplained Singularity</h2>
  ...
  <h2>Runaway Climate Change</h2>
  ...
  <h2>Global Epidemic</h2>
  ...
  <div class="Footer">
    ...
  </div>
</body>
```

Схема этой простой структуры выглядит таким образом:

```
1. How the World Could End
    1. Mayan Doomsday
    2. Robot Takeover
    3. Unexplained Singularity
    4. Runaway Climate Change
    5. Global Epidemic
```

Двухуровневая иерархия заголовков (<h1> и <h2>) отображается двухуровневой схемой. Эта схема похожа на схемы, которые создают средства отображения структуры документа многих текстовых редакторов, например MS Word 2010.

Еще один пример разметки:

```
<h1>Заголовок уровня 1</h1>
<h2>Заголовок уровня 2</h2>
<h2>Заголовок уровня 2</h2>
<h3>Заголовок уровня 3</h3>
<h2>Заголовок уровня 2</h2>
```

Дает следующую схему:

```
1. Заголовок уровня 1
    1. Заголовок уровня 2
    2. Заголовок уровня 2
        1. Заголовок уровня 3
    3. Заголовок уровня 2
```

Опять же, никаких сюрпризов здесь.

Алгоритм для создания схем достаточно умный и игнорирует пропущенные уровни. Например, в слегка корявой разметке, представленной далее, пропущен один уровень при переходе от заголовка первого уровня к заголовку третьего уровня:

```
<h1>Заголовок уровня 1</h1>
<h2>Заголовок уровня 2</h2>
<h1>Заголовок уровня 1</h1>
<h3>Заголовок уровня 3</h3>
<h2>Заголовок уровня 2</h2>
```

Схема этой разметки следующая:

```
1. Заголовок уровня 1
    1. Заголовок уровня 2
2. Заголовок уровня 1
    1. Заголовок уровня 3
    2. Заголовок уровня 2
```

Здесь заголовок третьего уровня находится на втором месте в схеме, по причине его позиции в документе. Это может показаться одним из случаев автоматического исправления ошибок, которое так любят делать браузеры, но в действительности такое поведение способствует достижению практической цели. В некоторых ситуациях веб-страница может быть собрана из нескольких частей, например, в нее может быть вставлена копия статьи, опубликованной где-то в другом месте. В таком случае уровни заголовков вставленного содержимого могут не совпадать с уровнями остального содержимого веб-страницы. Но так как алгоритм отображения схем устраняет такие несоответствия, то, скорее всего, проблем не будет.

Элементы для создания блоков

Элементы для создания блоков применяются для формирования новых, вложенных схем внутри страницы. Это элементы `<article>`, `<aside>`, `<nav>` и `<section>`. Чтобы понять, как работают элементы для создания блоков, представьте себе страницу, содержащую два элемента `<article>`. Так как `<article>` является элементом для создания блоков, такая страница будет иметь, по крайней мере, три схемы: схему всей страницы и по одной вложенной схеме для каждой из статей.

Чтобы лучше представить себе эту структуру, рассмотрим структуру статьи об апокалипсисе после ее приведения к стандарту HTML5:

```
<body>
  <article>
    <header>
      <h1>How the World Could End</h1>
      ...
    </header>

    <div class="Content">
      ...
      <h2>Mayan Doomsday</h2>
      ...
      <h2>Robot Takeover</h2>
      ...
      <h2>Unexplained Singularity</h2>
      ...
      <h2>Runaway Climate Change</h2>
      ...
      <h2>Global Epidemic</h2>
      ...
    </div>
  </article>

  <footer>
    ...
  </footer>
</body>
```

Вставьте эту разметку в какой-либо построитель схем, например на сайте **http://gsnedders.html5.org/outliner**, и вы увидите следующую схему:

```
1. Untitled Section
    1. How the World Could End
        1. Mayan Doomsday
        2. Robot Takeover
        3. Unexplained Singularity
        4. Runaway Climate Change
        5. Global Epidemic
```

В данном случае первый элемент схемы — это блок без заголовка, которым является корневой элемент `<body>`. Элемент `<article>` начинает новую, вложенную схему, которая содержит один элемент `<h1>` и несколько элементов `<h2>`.

Иногда наличие блока без заголовка "Untitled Section" указывает на ошибку. Хотя элементы `<aside>` и `<nav>` без заголовков считаются приемлемыми, такая терпимость обычно не распространяется на элементы `<article>` и `<section>`. В предыдущем примере блок без заголовка является основным блоком страницы, который принадлежит элементу `<body>`. Так как страница содержит одну статью, в отдельном заглавии для страницы нет смысла, поэтому можно не обращать внимание на эту странность.

Теперь посмотрим, что будет в случае более сложного примера, в качестве которого выступает веб-страница с боковой панелью на рис. 2.6. Построитель схем выдает для нее следующую схему:

```
1. Apocalypse Today
   1. Untitled Section
      1. Articles
      2. About Us
   2. How the World Could End
      1. Mayan Doomsday
      2. Robot Takeover
      3. Untitled Section
      4. Unexplained Singularity
      5. Runaway Climate Change
      6. Global Epidemic
```

В данном случае мы имеем два элемента для создания блоков и две вложенные схемы: одну — для боковой панели, а другую — для статьи. Также имеются два блока без названий, оба вполне законные. Первый из них — это элемент `<aside>` для боковой панели, а второй — элемент `<aside>` для броской цитаты во врезке в статье.

ПРИМЕЧАНИЕ

Кроме элементов для создания блоков существуют элементы, называющиеся *корнями блоков*. Эти элементы не просто ответвления какой-либо схемы, они начинают новую, собственную, схему, которая не входит в основную схему содержащей их страницы. Элемент `<body>`, в который заключено содержимое веб-страницы, является корнем блока, и это логично. Но в HTML5 корнями блоков считаются также элементы `<blockquote>`, `<td>`, `<fieldset>`, `<figure>` и `<details>`.

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ

Чем элементы для создания блоков полезны в случае сложных страниц?

Возможность разбиения содержимого на блоки очень полезна при *синдикации* и *агрегировании* — двух из нескольких способов публикации в веб-странице содержимого из других веб-сайтов.

Возьмем, например, веб-страницу, содержащую отрывки из нескольких статей, которые все размещены на других сайтах. Теперь представим, что эта страница имеет

структуру заголовков с большой глубиной вложения и где-то внутри этой иерархии заголовков, скажем под заголовком <h4>, размещена статья, взятая с другой веб-страницы.

В традиционной разметке HTML желательно, чтобы первый заголовок этого содержимого был уровня <h5>, т. к. он вложен под уровнем <h4>. Но эта статья первоначально была создана для размещения в другом месте, на другой странице с меньшей глубиной вложения блоков, поэтому она, вероятно, начинается с заголовка <h2> или даже <h1>. Это обстоятельство не мешает отображению страницы должным образом, но означает, что ее иерархия смешана, и страница окажется более трудной для обработки программами для чтения экрана, поисковыми движками и другим программным обеспечением.

Применение в разметке такой страницы элементов HTML5 решает эту проблему. Для этого вложенная статья вставляется в элемент <article>, вследствие чего содержимое из внешнего источника становится частью собственной схемы вложений. Эта схема может начинаться с заголовка любого уровня, для схемы страницы это не имеет никакого значения. Важным является его позиция в содержащем документе. Таким образом, если этот элемент <article> находится после заголовка уровня <h4>, тогда заголовок первого уровня внешней статьи ведет себя как заголовок логичного уровня <h5>, заголовок второго уровня — как заголовок логичного уровня <h6>, и т. д.

Из этого следует такой вывод: стандарт HTML5 предоставляет логическую систему создания схем, которая облегчает объединение документов. В этой системе создания схем важной является ***позиция заголовков***, а не их уровень, явно указанный в содержащих их элементах, т. е. <h1>, <h2>, <h3> и т. д. Это уменьшает вероятность допущения ошибок при разборе страницы.

Решение проблемы со схемой

Схемы веб-страниц во всех рассмотренных ранее примерах имеют полностью логическую структуру. Но в реальной жизни не все так просто, и могут возникать проблемы. Допустим, например, что мы создали следующий документ:

```
<body>
  <article>
    <h1>Чудеса природы, которые следует увидеть в своей жизни</h1>
    ...
    <h2>В Северной Америке</h2>
    ...
    <h3>Большой каньон</h3>
    ...
    <h3>Йеллоустонский национальный парк</h3>
    ...
    <h2>В остальном мире</h2>
    ...
    <aside>...</aside>
    ...
    <h3>Галапагосские острова</h3>
    ...
    <h3>Швейцарские Альпы</h3>
    ...
  </article>
</body>
```

Кажется, следует ожидать, что схема этой структуры будет следующей:

```
1. Untitled Section for the <body>
   1. Чудеса природы, которые следует увидеть в своей жизни
      1. В Северной Америке
         1. Большой каньон
         2. Йеллоустонский национальный парк
      2. В остальном мире
      3. Untitled Section for the <aside>
         1. Галапагосские острова
         2. Швейцарские Альпы
```

Но в действительности получим следующую схему:

```
1. Untitled Section for the <body>
   1. Чудеса природы, которые следует увидеть в своей жизни
      1. В Северной Америке
         1. Большой каньон
         2. Йеллоустонский национальный парк
      2. В остальном мире
      3. Untitled Section for the <aside>
      4. Галапагосские острова
      5. Швейцарские Альпы
```

Каким-то образом добавление элемента `<aside>` после элемента `<h2>` нарушает иерархию следующих элементов `<h3>`, делая их логический уровень таким же, как и уровень заголовка `<h2>`. Это, очевидно, не то, что нам требуется.

Чтобы решить это проблему, нужно сначала понять, что система HTML5 для создания схемы документа автоматически создает новый блок для каждого найденного ею элемента пронумерованного заголовка (т. е. `<h1>`, `<h2>`, `<h3>` и т. д.), *но не тогда*, когда этот элемент уже находится вверху блока.

В данном случае система создания схем не обращает внимания на начальный элемент `<h1>`, т. к. он находится вверху блока `<article>`. Но для следующих за ним элементов `<h2>` и `<h3>` новые блоки создаются. В результате получается схема, как будто бы была создана следующая разметка:

```
<body>
  <article>
    <h1>Чудеса природы, которые следует увидеть в своей жизни</h1>
    ...
    <section>
      <h2>В Северной Америке</h2>
      ...
      <section>
        <h3>Большой каньон </h3>
        ...
      </section>
```

```
      <section>
        <h3>Йеллоустонский национальный парк </h3>
        ...
      </section>
    </section>

    <section>
      <h2>В остальном мире</h2>
      ...
    </section>
    <aside>...</aside>
    ...
    <section>
      <h3>Галапагосские острова</h3>
      ...
    </section>
    <section>
      <h3>Швейцарские Альпы</h3>
      ...
    </section>
  </article>
</body>
```

В большинстве случаев эти автоматически создаваемые блоки не являются проблемой. Более того, эта особенность обычно полезна, т. к. обеспечивает помещение неверно пронумерованных заголовков на правильные уровни схемы *(см. врезку "Малоизвестная или недооцененная возможность. Чем элементы для создания блоков полезны в случае сложных страниц?" ранее в этой главе)*. Цена за это удобство — случайные сбои, подобные только что рассмотренному случаю.

Как можно видеть в этом листинге, поначалу все идет, как следует. Верхний элемент `<h1>` остается (т. к. он уже находится в элементе `<article>`), создается блок для первого элемента `<h2>`, потом вложенный блок для каждого элемента `<h3>` и т. д. Проблема возникает, когда процесс доходит до элемента `<aside>`. Этот элемент рассматривается как сигнал для закрытия текущего блока и означает, что блоки, создаваемые для следующих элементов `<h3>`, будут на таком же логическом уровне, что и предшествующие элементы `<h2>`.

Чтобы решить эту проблему, нужно взять под контроль создание блоков и подблоков, определив некоторые из них самостоятельно. В данном примере целью является предотвращение слишком раннего закрытия второго блока `<h2>`. Это можно сделать, явно определив его в разметке:

```
<body>
  <article>
    <h1>Чудеса природы, которые следует увидеть в своей жизни</h1>
    ...
    <h2>В Северной Америке</h2>
    ...
```

```
    <h3>Большой каньон</h3>
    ...
    <h3>Йеллоустонский национальный парк</h3>
    ...
    <section>
      <h2>В остальном мире</h2>
      ...
      <aside>...</aside>
      ...
      <h3>Галапагосские острова</h3>
      ...
      <h3>Швейцарские Альпы</h3>
      ...
    </section>
  </article>
</body>
```

Теперь алгоритм создания схем не обязан автоматически создавать блок для второго элемента <h2>, вследствие чего нет опасности, что он закроет этот блок, когда обнаружит элемент <aside>. Хотя можно было бы определить блок для каждого заголовка в этом документе, нет смысла загромождать этим разметку, т. к. всего лишь одно исправление решает проблему.

Примечание

Другой вариант решения — использовать вместо элемента <aside> элемент <div>. Так как <div> не является элементом создания блоков, он не вызовет преждевременное закрытие блока.

Использование элемента <aside> не всегда вызывает эту проблему. В предыдущих примерах этот элемент использовался для броской цитаты во врезке, но работал без проблем, т. к. располагался между двумя элементами <h2>. Но при помещении в разметке элемента создания блоков между двумя элементами заголовков разного уровня нужно проверить, не нарушена ли логика соответствующей схемы.

Совет

Не стоит слишком переживать, если вся эта концепция схем веб-страниц кажется слишком теоретизированной. По правде говоря, это довольно тонкая концепция, которую многие веб-разработчики будут игнорировать (по крайней мере, на данный момент). Лучше всего рассматривать систему HTML5 для создания схемы документа как полезный инструмент контроля качества. Просмотр схемы документа в одном из инструментов, описанных в *разд. "Как просмотреть схему веб-страницы" ранее в этой главе*, способен помочь уловить ошибки, которые могут указывать на другие проблемы, а также обеспечить правильное использование семантических элементов.

3

ГЛАВА 3

Разметка со смыслом

В предыдущей главе мы познакомились с семантическими элементами HTML5. С их помощью создаваемые веб-страницы можно аккуратно логически структурировать и подготовиться к появлению будущих сверхинтеллектуальных браузеров, поисковым движкам и вспомогательным средствам для просмотра страниц пользователями с ограниченными возможностями.

Но мы еще не закончили рассмотрение темы семантики веб-страниц. Семантика предназначена придать смысл разметке веб-страниц с помощью вставки в нее информации нескольких разных типов. В *главе 2* мы изучили разработку структуры страницы, применяемой для объяснения больших блоков содержимого и целых разделов разметки. Но кроме семантики блочного уровня, также существует семантика текстового уровня, добавляемая в разметку с целью объяснения единиц содержимого намного меньшего размера. Семантику текстового уровня можно использовать для того, чтобы указать на важную информацию, которая в противном случае была бы потеряна в море разметки веб-страницы. Такой информацией могут быть имена людей, адреса, календари событий, описания товаров, кухонные рецепты, обозрения разных типов и т. п. Обозначенную таким образом информацию потом могут извлекать и использовать разнообразные службы, начиная от интеллектуальных подключаемых модулей браузера до специализированных поисковых движков.

В этой главе мы сначала рассмотрим небольшой набор семантических элементов текстового уровня, встроенных в язык HTML5, которые можно без труда использовать в настоящее время. Далее мы рассмотрим сопутствующие стандарты, позволяющие работать с семантикой текстового уровня напрямую. Для этого нам придется разобраться с концепцией микроданных, которая возникла как часть первоначальной спецификации HTML5, но в настоящее время является отдельным, продолжающимся развиваться стандартом, ответственной за который является организация W3C. Попутно мы взглянем на несколько "дальновидных" служб, которые успешно используют микроданные уже в настоящее время.

Повторение семантических элементов

Мы начали рассматривать семантику с изучения элементов уровня структуры страницы потому, что разобраться с семантикой этого уровня довольно легко. А легкая она оттого, что структура подавляющего большинства веб-сайтов сравнительно одинаковая и создается с помощью небольшого набора элементов дизайна — верхних и нижних колонтитулов, боковых панелей и меню. Краткий обзор этих элементов приводится в табл. 3.1.

Таблица 3.1. *Семантические элементы блочного уровня*

Элемент	Описание
<article>	Представляет любую единицу информации, которую можно рассматривать, как статью — блок самодостаточного содержимого наподобие газетной статьи, сообщения на форуме или записи в блоге (исключая второстепенную информацию, такую как комментарии или биография автора)
<aside>	Представляет цельный фрагмент содержимого, отдельный от окружающего его остального содержимого страницы. Например, логично использовать элемент <aside>, чтобы создать боковую панель, содержащую связанное содержимое или ссылки на основную статью
<figure> и <figcaption>	Представляет рисунок. Элемент <figcaption> содержит текст подписи к рисунку, а элемент <figure> — элементы <figcaption> и <img>, который собственно и ссылается на изображение. Цель такой комбинации — указать связь между рисунком и его подписью
<footer>	Представляет нижний колонтитул страницы. Это небольшой фрагмент содержимого, который может включать "мелкий текст", сообщение об авторских правах и несколько ссылок (например, "О нашей компании" или "Поддержка")
<header>	Представляет расширенный верхний колонтитул, включающий стандартный заголовок HTML и дополнительное содержимое. Дополнительное содержимое может состоять из логотипа, имени автора и набора навигационных ссылок для основного содержимого страницы
<hgroup>	Представляет расширенный заголовок, содержащий два или более пронумерованных элемента заголовка (т. е. <h1>, <h2>, <h3> и т. д.), и ничего больше. Основное назначение этого элемента — держать заголовок и подзаголовок вместе
<nav>	Представляет значительный набор ссылок на странице, указывающих на тематические разделы текущей страницы или на другие страницы веб-сайта. На практике страницы с несколькими блоками <nav> не являются редкостью
<section>	Представляет раздел документа или группу документов. Элемент <section> является универсальным контейнером, использование которого регламентируется единственным правилом: его содержимое должно начинаться с заголовка. Элемент следует использовать только в тех случаях, когда не подходят никакие другие семантические элементы (такие как, например, <article> или <aside>)

Семантика текстового уровня намного сложнее по причине огромного количества типов содержимого. Если бы HTML5 содержал элемент для каждого типа инфор-

мации, которую можно включить в веб-страницу, то вместо четкой и ясной разметки такое изобилие семантических элементов, наоборот, сделало бы ее более запутанной. Проблема усложняется тем, что структурированная информация также состоит из более мелких составных частей, которые можно компоновать разными способами. Например, даже для обычного почтового адреса потребовалась бы целая куча семантических элементов (наподобие `<address>`, `<name>`, `<street>`, `<postalcode>`, `<country>` и т. д.), прежде чем его можно было использовать в разметке страницы.

Для решения этой проблемы в HTML5 применяются два подхода. Во-первых, вводится очень небольшое количество семантических элементов текстового уровня. Во-вторых, что более важно, HTML5 поддерживает отдельный стандарт микроданных, который предоставляет разработчикам расширенный способ определить по своему желанию любой тип информации, а потом пометить эту информацию в разметке флагами. В этой главе рассматриваются оба эти подхода. Начнем их рассмотрение с трех новых семантических элементов текстового уровня: `<time>`, `<output>` и `<mark>`.

Обозначение дат и времени с помощью элемента <*time*>

Веб-страницы часто содержат информацию о датах и времени. Например, такая информация вставляется в конце большинства записей в блогах. К сожалению, стандартного способа помечать даты не существует, вследствие чего другим программам (например, поисковым движкам) не так-то легко извлечь их из остального кода разметки. Эта проблема решается введением семантического элемента `<time>`, с помощью которого можно обозначить дату, время или комбинацию обоих. В следующем коде приводится пример использования этого элемента:

```
Новый магазин откроется <time>2012-03-21/time>.
```

> ***Примечание***
>
> То, что элемент `<time>` применяется как оболочка для даты (без времени), может показаться несколько нелогичным, но это просто одна из странностей HTML5. Логичнее было бы ввести элемент `<datetime>`, но этого почему-то не сделали.

От элемента `<time>` требуется выполнять две функции. Во-первых, он должен указывать местонахождение значения даты или времени в разметке. Во-вторых, он должен предоставлять заключенное в него значение даты или времени в форме, понимаемой любой программой. Предыдущий пример отвечает второму требованию предоставления универсального формата даты, который состоит из четырехсимвольного значения года, двухсимвольного значения месяца и двухсимвольного значения дня, в указанном порядке. Иными словами, формат даты имеет такой шаблон:

```
ГГГГ:ММ:ДД
```

Но пользователю, просматривающему веб-страницу, дату вполне допустимо показать в другом формате. Более того, дату на экран можно выводить любым нравя-

щимся вам способом, при условии предоставления ее значения в машиночитаемом универсальном формате в атрибуте `datetime`, как показано в следующем примере:

```
Новый магазин откроется <time datetime="2012-03-21">21-го марта</time>.
```

В браузере это будет выглядеть таким образом:

```
Новый магазин откроется 21-го марта.
```

Подобные правила также применяются и для значения времени, которое предоставляется в таком формате:

```
ЧЧ:ММ+00:00
```

То есть, дается двузначное значение часа (в 24-часовом формате), затем двузначное значение минут. Часть после знака "плюс" (или "минус") представляет смещение часового пояса от всемирного координированного времени. Указание смещения часового пояса является обязательным. Узнать смещение часового пояса для конкретного региона можно на веб-сайте **http://en.wikipedia.org/wiki/Time_zone**. Например, Москва находится в московском часовом поясе, который также называется UTC+4:00[1], т. е. смещение данного часового пояса равно плюс четырем часам. Время 16:30 в Москве указывается в разметке следующим образом:

```
Магазин открывается ежедневно в <time datetime="16:30+4:0">16:30</time>.
```

Таким образом, посетителям веб-страницы время представляется в привычном для них формате, в то время как для поисковых роботов и другого программного обеспечения предоставляется однозначное значение `datetime`, с которым они могут работать.

Наконец, время в определенный день указывается путем совмещения этих двух стандартов. Дата приводится первой, за ней следует прописная латинская буква T, после которой указывается время:

```
Новый магазин откроется <time datetime="2012-03-21T16:30+4:00">
                                            21-го марта в 16:30</time>.
```

Элемент `<time>` также поддерживает атрибут `pubdate`. Он применяется в тех случаях, когда указываемая дата совпадает с датой публикации текущего содержимого (например, статьи, в которой помещается элемент `<time>`). В качестве примера можно привести такой код:

```
Опубликовано <time datetime="2011-03-21" pubdate>21-го марта 2011 г.</time>
```

Примечание

Так как элемент `<time>` является чисто информационным и не обеспечивает никакого связанного с ним форматирования, его можно использовать для любого браузера, не волнуясь о совместимости. Но для форматирования этого элемента необходимо прибегать к обходному решению для Internet Explorer, описанному в *разд. "Браузерная совместимость для семантических элементов" главы 2*.

---

[1] UTC (Universal Time, Coordinated) — всемирное координированное время.

JavaScript-вычисления и элемент *<output>*

Семантический элемент <output> предназначен для облегчения понимания разметки определенных страниц, в которых используются сценарии JavaScript. Функция этого элемента состоит просто в резервировании поля подстановки, в которое код может выводить результаты вычислений.

Допустим, например, что вы создали страницу наподобие показанной на рис. 3.1.

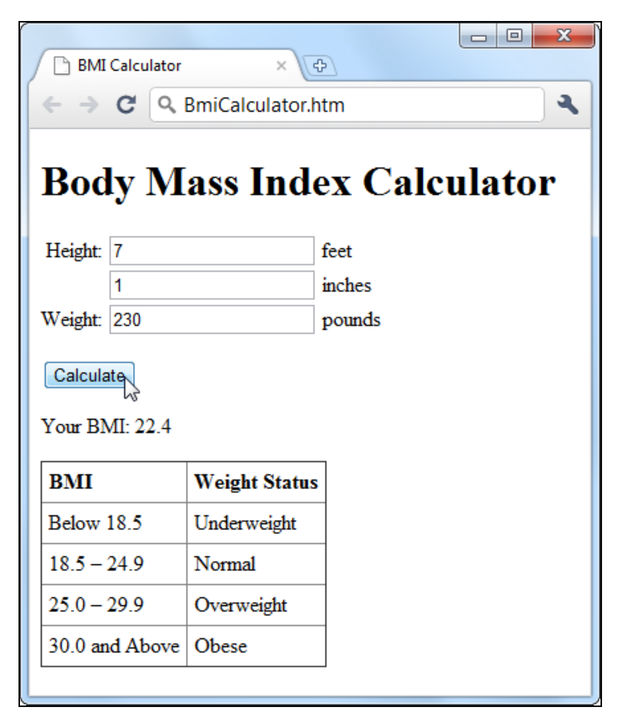


Рис. 3.1. Проверенный временем шаблон для онлайновых вычислений. Введите требуемые данные, нажмите кнопку, и форма выводит результат

Показанная на рис. 3.1. форма разрешает пользователю ввести определенную информацию в соответствующие поля. Нажатие кнопки **Calculate** отправляет эту информацию сценарию JavaScript, который выполняет необходимые вычисления и выводит результаты этих вычислений в соответствующих полях внизу формы.

В таких случаях для обозначения места для вывода результатов полю подстановки присваивается однозначный идентификатор. Обычно для поля подстановки используется элемент <span>, который отлично работает, но не придает этому полю никакого специфического значения:

```
<p>Your BMI: <span id="result"></span></p>
```

Использование же элемента <output> для этой цели делает код более осмысленным:

```
<p>Your BMI: <output id="result"></output></p>
```

Собственно код JavaScript не требует никаких изменений, т. к. он ищет элемент по его идентификатору, и его ничуть не волнует тип этого элемента:

```
var resultElement = document.getElementById("result");
```

> **ПРИМЕЧАНИЕ**
>
> Прежде чем использовать элемент `<output>` с Internet Explorer, обязательно включите в свой код обходное решение для этого браузера, описанное в *разд. "Браузерная совместимость для семантических элементов" главы 2*. В противном случае, на старых версиях IT этот элемент будет недоступным в JavaScript.

Часто для удобства элементы управления помещаются в элемент `<form>`. В следующем листинге показан соответствующий код, в котором определяются три текстовых поля для ввода информации:

```
<form action="#" id="bmiCalculator">
  <label for="feet inches">Height:</label>
  <input name="feet"> feet<br>
  <input name="inches"> inches<br>

  <label for="pounds">Weight:</label>
  <input name="pounds"> pounds<br><br>
  ...
</form>
```

Элемент `<output>` можно сделать еще более осмысленным, добавив в него атрибут `form` (который указывает идентификатор формы, содержащей связанные элементы управления) и атрибут `for` (со списком идентификаторов связанных элементов управления, разделенных запятыми). Далее показан примера такого кода:

```
<p>Your EMI: <output id="result" form="bmiCalculator"
for="feet inches pounds"></output></p>
```

Эти атрибуты в действительности ничего не делают, кроме как передают информацию о том, откуда элемент `<output>` получает свои данные. Но они вносят очень важную информацию с точки зрения семантики. В частности, если страницу нужно будет отредактировать кому-то другому, а не разработчику-хозяину, эти атрибуты помогут ему разобраться, как она работает.

> **СОВЕТ**
>
> Если ваших знаний о формах недостаточно для понимания этой главы, *см. главу 4*. А если вы знаете эсперанто лучше, чем JavaScript, то можете освежить свои знания этого языка программирования, ознакомившись с *приложением 2*. Наконец, если вы хотите исследовать весь код примера, его можно загрузить с сайта **http://reboot-me.com/prosetech/html5**.

Выделение текста цветом с помощью элемента *<mark>*

Элемент `<mark>` обозначает блок текста, выделенного цветом. Применение этого элемента особенно подходит для выделения фрагментов текста, как показано в следующем листинге:

```
<p>In = 2009, Facebook made a bold grab to own everyone's
   content, <em>forever</em>. This is the text they put in their
   terms of service:</p> <blockquote>You hereby grant Facebook an
   <mark>irrevocable, perpetual, non-exclusive, transferable,
   fully paid, worldwide license</mark> (with the right to
   sublicense) to <mark>use, copy, publish</mark>, stream, store,
   retain, publicly perform or display, transmit, scan, reformat,
   modify, edit, frame, translate, excerpt, adapt, create
   derivative works and distribute (through multiple tiers),
   <mark>any user content you post</mark>
   ...
</blockquote>
```

Текст, взятый в элемент <mark>, выделяется желтым цветом, как показано на рис. 3.2.

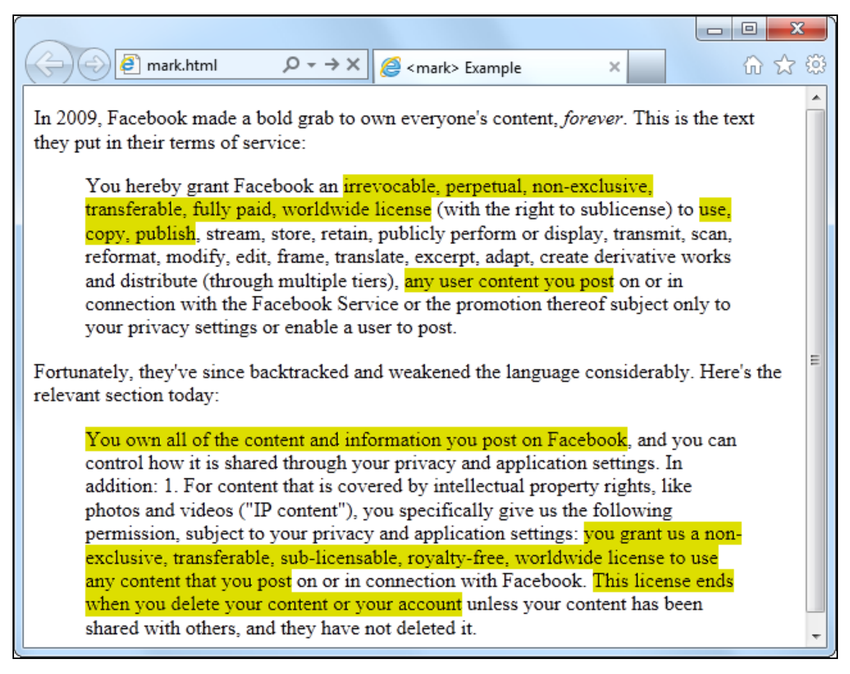


Рис. 3.2. Эффект использования элемента <mark> в разметке

С помощью элемента <mark> можно также помечать важное содержимое или ключевые слова (наподобие того, как это делают поисковые движки, выделяя текст, совпадающий с текстом запроса, в результатах поиска) или совместно с элементами <del> (удалено) и <ins> (вставлено) помечать изменения в документе.

По правде говоря, элемент <mark> немного выделяется из компании остальных семантических элементов. Хотя спецификация HTML5 считает его семантическим элементом, его презентационная функция, возможно, более важная, чем семантическая. По умолчанию текст, помеченный элементом <mark>, выделяется желтым цветом, но с помощью правил таблицы стилей можно применить какой-либо другой цвет.

Совет

По сути, элемент `<mark>` не предназначен для выполнения форматирования текста. В конце концов, имеется множество других способов выделить текст на веб-странице. Поэтому элемент `<mark>` следует использовать (совместно с форматированием посредством CSS, если есть такое желание), когда это подходит с точки зрения семантики. Удачно использовать элемент `<mark>` для привлечения внимания к обычному тексту, содержание которого по какой-либо причине является важным, например из-за контекста обсуждения, в котором он находится, или из-за выполняемой пользователем операции.

Даже если оставляется форматирование по умолчанию (черный текст на желтом фоне), следует добавить правило таблицы стилей для этого форматирования для браузеров, которые не поддерживают HTML5. Далее приводится пример такого правила:

```
mark {
  background-color: yellow;
  color:            black;
}
```

А чтобы форматирование элемента `<mark>` работало в Internet Explorer, нужно добавить обходное решение, описанное в *разд. "Браузерная совместимость для семантических элементов" главы 2*.

Другие стандарты, улучшающие семантику

Сейчас вы, наверное, думаете, что в разметке веб-страниц можно было использовать множество других семантических элементов, которых в HTML5 нет. Ну да, имеющимися элементами можно помечать даты и выделять цветом текст, но как насчет других единиц информации — адресов, названий компаний, описания товаров, персональных сведений и т. п.? Создатели HTML5 преднамеренно обошли эту область, т. к. не хотели загромождать язык десятками специализированных элементов, необходимых одним разработчикам, но полностью бесполезных для других. Чтобы по-настоящему перейти на следующий уровень в семантике, необходимо выйти за пределы базового языка HTML5 и рассмотреть некоторые другие стандарты, которые можно применять в разработке веб-страниц.

Идея семантически осмысленной разметки не является новой. В действительности, давным-давно, когда HTML5 существовал только в задумках Яна Хиксона (Ian Hickson), редактора группы WHATWG, многие веб-разработчики настойчиво искали методы для создания более осмысленной разметки. Не всегда их требования совпадали: одни хотели улучшить доступность своих веб-страниц, другие — возможность для интеллектуального анализа данных, а третьим просто хотелось повысить фактор впечатляемости своего резюме. Но никто из них не смог найти то, что им требовалось в стандартном языке HTML, вследствие чего было создано несколько новых стандартов, чтобы заполнить этот пробел.

В последующих разделах мы рассмотрим аж четыре таких стандарта. Сначала мы познакомимся со стандартом ARIA, предназначенным для улучшения доступности

веб-страниц для программ чтения экрана. Потом мы вкратце рассмотрим три конкурирующих подхода к описанию содержимого самых разнообразных типов, будь то подробная контактная информация, названия компаний или просто все что угодно, что можно вставить в теги HTML-страницы.

Стандарт Accessible Rich Internet Applications

Развивающийся стандарт Accessible Rich Internet Applications (ARIA, доступные активные веб-приложения) позволяет предоставлять дополнительную информацию для программ чтения экрана с помощью атрибутов любого элемента HTML. Среди прочих, в нем вводится атрибут `role`, который указывает назначения данного элемента. Например, назначение представляющего верхний колонтитул элемента `<div>`:

```
<div class="header">
```

можно довести до сведения программ чтения экрана, присвоив атрибуту `role` значение `banner`:

```
<div class="header" role="banner">
```

Но, опять же, как мы узнали в *главе 2*, стандарт HTML5 предоставляет нам более осмысленный способ обозначения верхних колонтитулов. Поэтому нам следует использовать что-то наподобие следующего:

```
<header role="banner">
```

Этот пример иллюстрирует два важных факта. Первый: стандарт ARIA требует использования одного из коротких списков рекомендованных названий ролей. (Полный список см. в соответствующем разделе спецификации по адресу **www.w3.org/TR/wai-aria/roles#landmark_roles**.) Второй: некоторые части стандарта ARIA перекрывают семантические элементы HTML5, что вполне логично, т. к. ARIA появился раньше HTML5. Но это не совсем полное перекрытие. Например, некоторые названия ролей дублируют элементы HTML5 (скажем, `banner` и `article`), в то время как другие являются более продвинутыми (например, `toolbar` и `search`).

Стандарт ARIA также вводит два атрибута для работы с HTML-формами. Атрибут `aria-required` в текстовом поле указывает, что пользователю нужно ввести значение. А атрибут `aria-invalid` указывает на недействительность текущего значения в текстовом поле. Польза от этих атрибутов в том, что программы чтения экрана могут пропустить визуальные подсказки, которыми руководствуются пользователи с нормальным зрением при заполнении форм, например, звездочку рядом с незаполненным полем или красный мигающий восклицательный знак рядом с полем с неправильным введенным значением.

Чтобы правильно применять стандарт ARIA, необходимо изучить его и потратить некоторое время на исследование разрабатываемой разметки. С учетом того, что стандарт ARIA все еще находится в процессе развития и HTML5 предоставляет некоторые такие же возможности, но меньшей ценой, мнение веб-разработчиков касательно ценности такого вложения усилий и времени в этот стандарт неоднознач-

но. Но если вы хотите создать веб-сайт по настоящему доступный уже сегодня, нужно использовать оба стандарта, т. к. новые программы чтения экрана поддерживаются стандарт ARIA, а стандарт HTML5 — еще нет.

ПРИМЕЧАНИЕ

Дополнительную информацию о стандарте ARIA (полное название которого — WAI-ARIA, т. к. он был разработан группой WAI) см. в его спецификации на сайте **www.w3.org/TR/wai-aria**.

Стандарт Resource Description Framework

Стандарт Resource Description Framework (RDFa, инфраструктура для описания ресурсов — в атрибутах) определяет правила для интегрирования подробных метаданных в веб-документы посредством атрибутов. У стандарта RDFa есть значительное преимущество перед другими подходами, рассматриваемыми в этой главе: это стабильный, установившийся стандарт. Но у него также два значительных недостатка. Первый — это сложный стандарт. Разметка со вставленными в нее метаданными RDFa получается намного объемистей и существенно неуклюжей, чем обычная HTML-разметка. Второй — стандарт этот разработан для применения с XHTML, а не с HTML5. В настоящее время несколько сверхинтеллектуальных веб-гуру занимается разработкой лучших способов адаптирования RDFa для работы с HTML5. Но, возможно, что RDFa просто не приживется в мире HTML5, т. к. ему больше подходят строгий синтаксис и железные правила XML.

В этой главе стандарт RDFa не рассматривается. Но если вы хотите узнать больше о нем, солидный обзор можно получить на сайте **http://en.wikipedia.org/wiki/RDFa**. А на веб-сайте Google Rich Snippets *(см. разд. "Расширенные фрагменты страницы" далее в этой главе)* можно ознакомиться с версиями RDFa для всех примеров расширенных фрагментов Google.

Микроформаты

Микроформаты предоставляют простой, прямолинейный подход к внедрению метаданных в веб-страницы, не придерживаются какого-либо определенного официального стандарта и не являются таковым. Лучше всего микроформаты можно описать как несвязанный набор соглашений, которые позволяют веб-страницам предоставлять структурированную информацию, не требуя сложных стандартов, наподобие RDFa. Благодаря этому подходу микроформаты пользуются огромным успехом; в недавнем исследовании, проведенном Google, было обнаружено, что в 94% страниц, содержащих какой-либо тип расширенных метаданных, эти метаданные предоставляются микроформатами.

ПРИМЕЧАНИЕ

Если судить на основе популярности микроформатов, то можно подумать, что исход битвы за семантический Интернет уже решен. Такое заключение будет преждевременным по нескольким причинам. Прежде всего, подавляющее большинство веб-страниц не содержит расширенных семантических данных вообще. Вдобавок в боль-

шинстве веб-страниц, применяющих микроформаты, это делается только в двух целях: для обозначения контактной информации и событий. И наконец, простота микроформатов может удерживать их от применения для более продвинутых задач, особенно когда уровень возможности HTML5 сравняется с их уровнем. Поэтому, хотя микроформаты никуда в ближайшее время не исчезнут, игнорировать их конкурентов тоже не стоит.

Прежде чем помечать данные микроформатом, нужно выбрать, какой именно микроформат использовать для этого. Широко применяется всего лишь пара десятков микроформатов, большинство из которых продолжает совершенствоваться и обновляться. Информацию о доступных микроформатах и подробное описание каждого из них можно найти на веб-сайте **http://microformats.org/wiki/Main_Page-ru**. Наибольшей популярностью среди всех этих микроформатов пользуются два: hCard и hCalender, которые мы и рассмотрим в следующих двух разделах.

Обозначение контактной информации с помощью микроформата hCard

Микроформат hCard предоставляет универсальный способ обозначения контактной информации для физического лица, компании или места. По результатам последнего подсчета Интернет содержит свыше 2 млрд hCard, что делает этот микроформат безоговорочно самым популярным.

Микроформаты работают по инновационному методу — они добавляют свою информацию поверх атрибута `class`, который обычно используется для форматирования. Данные помечаются, используя определенные стандартные названия стилей, на основе типа данных. Потом такую разметку может прочитать другая программа, извлечь помеченные данные и по атрибутам определить их значение. Для создания hCard требуется корневой элемент, который присваивает атрибуту `class` значение `vcard`. Внутри этого элемента необходимо посредством другого элемента предоставить, по крайней мере, отформатированное имя. Атрибуту `class` этого внутреннего элемента нужно присвоить значение `fn`. Вот приводится пример такого определения:

```
<div class="vcard">
  <div class="fn">Mike Rowe Formatte</div>
</div>
```

При использовании для микроформата атрибута `class` в создании соответствующего стиля с именем этого класса нет надобности, более того, это, скорее всего, только создаст путаницу. Вместо этого атрибуту `class` дается другое задание — объявлять, что его содержимое является хорошо структурированными, осмысленными данными.

Хотя единственной обязательной единицей информации для микроформатов hCard является имя, большинство из них содержит дополнительные подробности, такие как почтовый адрес и адрес электронной почты, URL веб-сайта, номер телефона, дата рождения, фотография, должность, название организации и т. п. При условии использования правильных названий классов все эти подробности можно вставить

в элемент, содержащий класс `vcard`. В следующем листинге приводится пример определения hCard, содержащего разные типы персональной информации:

```
<div class="vcard">
  <div class="fn">Mike Rowe Formatte</div>
  <img class="photo" src="face.jpg" alt="Mike's Face">
  <div class="title">Web Developer</div>
  <div class="org">The Magic Semantic Company</div>
  <a class="url" href="http://www.magicsemantics.com">
                                     www.magicsemantics.com</a>
  <div class="tel">641-545-0234</div>
</div>
```

Совет

Информацию обо всех свойствах hCard см. на веб-сайте **http://microformats.org/wiki/hcard-ru**.

В предыдущем примере мы рассмотрели, как создать hCard с самого начала, но микроформатом часто требуется пометить данные в уже существующих веб-страницах. Например, может потребоваться пометить форматом hCard контактные данные веб-страницы, чтобы сделать их доступными любой программе, понимающей этот микроформат. Это довольно простая задача, при условии следования нескольким правилам.

- Часто важные данные будут смешаны с содержимым, которое не требуется выделять. В таком случае можно заключить в новый элемент каждую единицу информации, которую нужно выделить. Для содержимого блочного уровня используется элемент `<div>`, а для текстового — элемент `<span>`.
- Не беспокойтесь о других элементах с разными названиями классов. При обработке микроформата браузер игнорирует все, что не имеет распознаваемого названия класса.
- Для обозначения микроформатом изображения можно использовать элемент `<img>`, а для ссылки — элемент `<a>`. В остальных случаях форматом будет, скорее всего, обозначаться простой текст.

Рассмотрим типичный пример обозначения микроформатом данных в существующей странице. Допустим, у нас есть страница "About Me", содержащая контактную информацию создателя веб-сайта (рис. 3.3).

Обычная HTML-разметка этой страницы будет подобна следующей:

```
<h1>About Me</h1>

<img src="face.jpg" alt="Mike's Face">
<p>This website is the work of <b>Mike Rowe Formatte</b>.
His friends know him as <b>The Big M</b>.</p>

<p>You can contact  him where he works, at
The Magic Semantic Company (phone
641-545-0234 and ask for Mike).</p>
```

```
<p>Or, visit him there at:<br>
42 Jordan Gordon Street, 6th Floor<br>
San Francisco, CA 94105<br>
USA<br>
<a href="http://www.magicsemantics.com">www.magicsemantics.com</a>
```

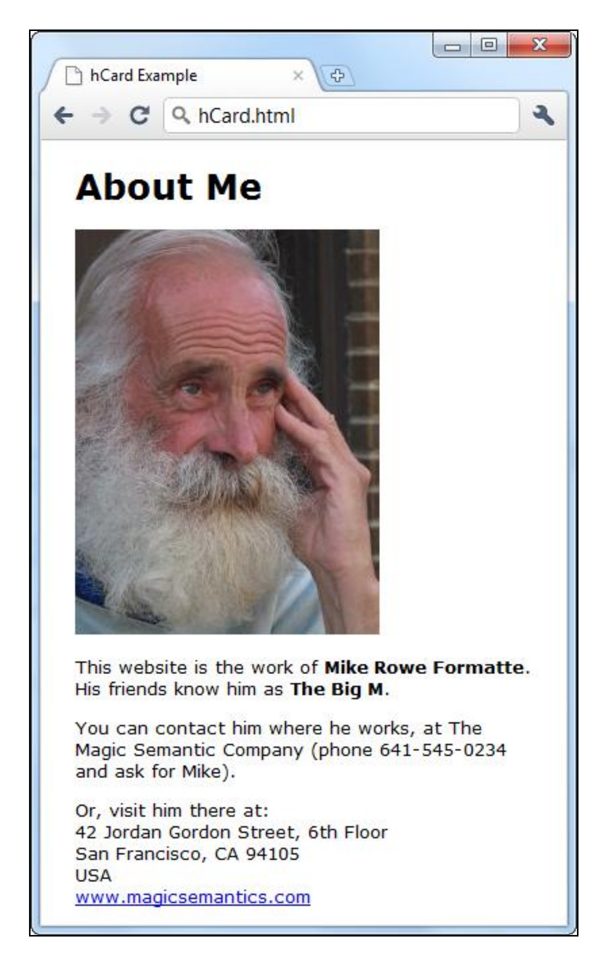


Рис. 3.3. Страница "About Me" с контактной информацией создателя веб-сайта

Добавив в эту разметку соответствующее микроформатирование, получим следующий код:

```
<h1>About Me</h1>

<div class="vcard">
  <img class="photo" src="face.jpg" alt="Mike's Face">
  <p>This website is the work of
  <span class="title" style="display:none">Web Developer</span>
  <b class="fn">Mike Rowe Formatte</b>.
  His friends know him as <b class="nickname">The Big M</b>.</p>
```

```
<p>You can contact him where he works, at
<span class="org">The Magic Semantic Company</span> (phone
<span class="tel">641-545-0234</span> and ask for Mike).</p>

<p>Or, visit him there at:<br>
<span class="street-address">42 Jordan Gordon Street,
                                        6th Floor</span><br>
<span class="locality">San Francisco</span>,
<span class="region">CA</span>
<span class="postal-code">94105</span><br>
<span class="country-name">USA</span><br>
<a class="url" href="http://www.magicsemantics.com">
                                  www.magicsemantics.com</ a>
</div>
```

Для вставки микроформата над исходной разметкой были выполнены следующие преобразования.

- Добавлено несколько новых элементов `<span>` в качестве оболочки для содержимого, требуемого классу `vcard`.
- Там, где это уместно, вместо элемента `<span>` добавлены атрибуты `class` в уже существующие элементы. Например, информация об имени уже вставлена в элемент `<b>`, поэтому нет надобности добавлять новый элемент `<span>`. (Конечно же, ничто не запрещает делать это. Например, если требуется одновременно пометить и отформатировать имя, может быть, предпочтительнее использовать что-то наподобие `<b class="Keyword Emphasis"><spanclass="fn">Mike Rows Formatte</span></b>`. Но в таком случае правила таблицы стилей следует держать отдельно от микроформата.)
- Для указания рода занятий лица (Web Developer) применяется скрытая единица содержимого. Показывать эту информацию на странице нет надобности, т. к. она уже очевидна из предложения "This website is the work of..." ("Этот веб-сайт создан"). Но этот метод еще надо обсудить, т. к. некоторые средства (например, Google) не обращают внимания на информацию в разметке, которая не предоставляется для просмотра посетителю страницы.

Теперь давайте рассмотрим, какой результат мы получим в ответ на все наши усилия, приложенные к оформлению микроформатом необходимых данных на странице. Хотя ни один из существующих браузеров не имеет естественной поддержки микроформатов (по крайней мере, на момент написания этой книги), существуют разнообразные подключаемые модули и сценарии, которые могут дать им эту возможность. Пользу от такой возможности не трудно вообразить. Например, браузер мог бы обнаружить элементы hCard на странице, вывести их список в боковой панели и предоставить пользователю команды, позволяющие добавить любое лицо из списка в личную адресную книгу пользователя так же быстро, как и добавить страницу в закладки. Информацию о подключаемых модулях браузера для поддержки микроформатов можно получить на веб-сайте **http://microformats.org/wiki/browsers**.

Одним из таких расширений является Oomph для Internet Explorer, которое можно загрузить с веб-сайта **http://oomph.codeplex.com**. Этот модуль автоматически исследует каждую посещаемую веб-страницу на предмет наличия в ней трех микроформатов: hCard (для контактной информации), hCalendar (для календарей событий) и hMedia (для изображений, аудио и видео). В случае обнаружения в странице данных, помеченных одним из этих форматов, в левом верхнем углу окна браузера выводится значок Oomp (рис. 3.4, *вверху*.) Щелчок по этому значку выводит в окно браузера всю обнаруженную информацию, а также элементы управления для ее обработки (рис. 3.4, *внизу*).

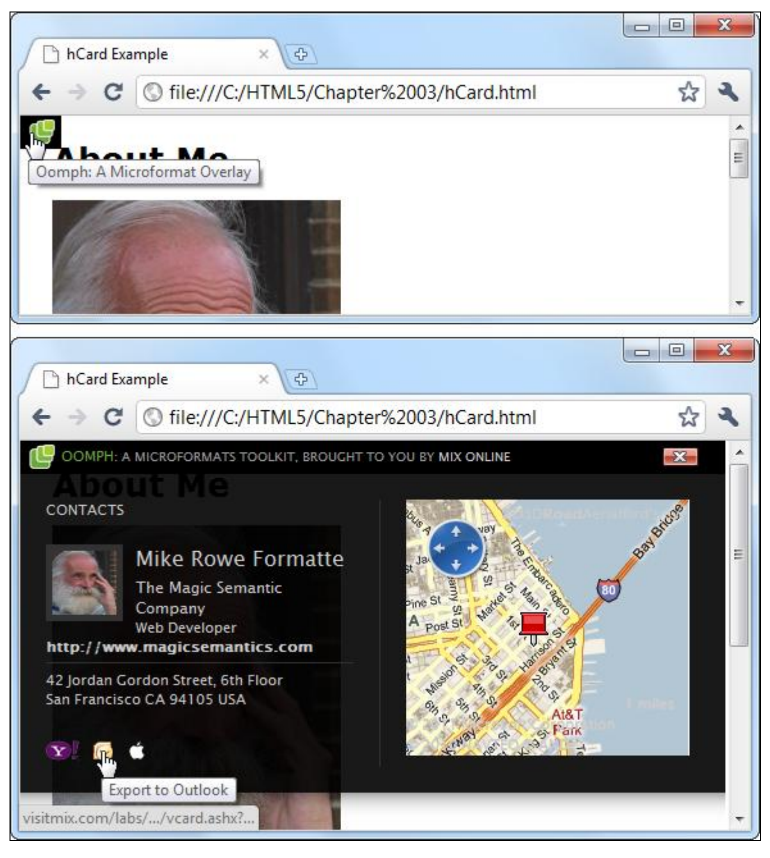


Рис. 3.4. *Вверху:* сигнальный значок указывает, что Oomph нашел, по крайней мере, одну единицу информации, помеченную микроформатом. *Внизу*: щелчок по значку выводит всю информацию, включая карту связанного географического местоположения и кнопку для переноса контактной информации в адресную книгу электронной почты. Если страница содержит несколько разных микроформатов, также выводятся ссылки, с помощью которых можно просмотреть все разделы hCard, hCalendar и hMedia

Но что особенно интересно в этом инструменте, так это возможность использовать его другим способом — через библиотеку JavaScript. Для этого всего лишь нужно добавить в страницу ссылки на сценарии:

```
<head>
  <meta charset="utf-8">
  <title>hCard Example</title>
```

```
  <script src="jquery-1.3.2.min.js"></script>
  <script src="oomph.min.js"></script>
  ...
</head>
```

Теперь все посетители вашей страницы получат возможность доступа к помеченным микроформатами данным страницы, независимо от используемого ими браузера. Надо сказать, что именно таким образом пример, показанный на рис. 3.4, работает в браузере Chrome. Вы также можете попробовать эту возможность в своем браузере, посетив наш сайт по адресу **http://www.prosetech.com/html5/** и щелкнув по ссылке **Microformats.html** в разделе **Chapter 3. Meaningful Markup**.

Примечание

Модуль Oomph следует рассматривать как демонстрацию принципиальной возможности реализации средств, использующих микроформаты, а не как инструмент для практического использования. Наилучшим шансом для будущего успеха микроформатов будет их прямая поддержка основными браузерами, точно так браузеры Internet Explorer, Firefox и Safari поддерживают каналы RSS в настоящее время. Например, при открытии страницы какого-либо блога в Firefox этот браузер автоматически обнаруживает канал RSS и предоставляет пользователю возможность создать "живую" закладку. Как раз эта дополнительная возможность сможет сделать микроформаты по настоящему полезными.

Обозначение событий с помощью микроформата hCalendar

Вторым по уровню популярности микроформатом является hCalendar, предоставляющий простой способ для разметки событий. Например, с помощью этого микроформата можно помечать встречи, собрания, праздники, выпуски продукции, открытия магазинов и т. п. В настоящее время в Интернете есть десятки миллионов событий, помеченных микроформатом hCalendar. Пример отображения одного такого события показан на рис. 3.5.

Если вы разобрались в том, как помечать информацию микроформатом hCard, у вас не будет никаких трудностей и с микроформатом hCalendar. Сначала событие нужно поместить в элемент с названием класса `vevent`. Внутри этого элемента нужно поместить, по крайней мере, две единицы информации: дату начала (которая помечается классом `dtstart`) и описание события (которое помечается классом `summary`). Дополнительно можно вставить разные необязательные атрибуты (описание которых см. на **http://microformats.org/wiki/hcalendar-ru**), включая конечную дату или длительность события, место его проведения, а также URL страницы, содержащей более подробную информацию о событии. Далее представлен пример кода для пометки события:

```
<div class="vevent">
  <h2 class="summary">Web Developer Clam Bake</h2>
  <p>I'm hosting a party!</p>
  <p>It's
  <span class="dtstart" title="2011-10-25T13:30">Tuesday,
                                        October 25, 1:30PM</span>
```

```
  at the <span class="location">Deep Sea Hotel, San
                                          Francisco, CA</span></p>
</div>
```

Дату следует форматировать согласно требованиям универсального формата данных *(см. разд. "Обозначение дат и времени с помощью элемента <time>" ранее в этой главе)*. Но есть и обходное решение — машиночитаемую информацию о дате можно предоставить с помощью атрибута `title`, а потом использовать любой, нравящийся вам текст. К сожалению, в настоящее время для микроформатов не существует автоматического способа использования информации в атрибуте `datetime` HTML5-элемента `<time>`. Но этот недостаток устранен в стандарте микроданных, который рассматривается в следующем разделе.



Рис. 3.5. Во всех страницах событий Facebook события помечаются микроформатом hCalendar, а место события — микроформатом hCard. Благодаря этому, с помощью расширения браузера посетители страницы могут записать эти данные на свой компьютер. (Для браузера Firefox можно воспользоваться расширением Operator, загрузить которое можно по адресу **http://addons.mozilla.org/firefox/addon/operator**.)

Микроданные

Еще одним подходом к решению задачи придания семантического смысла разметке веб-страниц являются микроданные. Формат микроданных зародился как часть спецификации HTML5, но потом был выделен в отдельный развивающийся стандарт (см. **http://dev.w3.org/html5/md**). В микроданных применяется подход, подобный стандарту RDFa, но проще. В отличие от микроформатов, микроданные имеют собственные атрибуты, что устраняет возможность конфликта с правилами таблиц

стилей (или озадачивания веб-разработчиков, пытающихся разобраться в чужом коде). Такой подход делает форматы микроданных более логичными, чем другие форматы, а также более приспосабливаемыми к использованию с языками собственной разработки других веб-дизайнеров. Но за это приходится платить ценой потери краткости — помеченная микроданными разметка может быть несколько большего объема, чем разметка, помеченная микроформатами.

ПРИМЕЧАНИЕ

Так как микроданные являются самым новым стандартом, пока еще не ясно, сможет ли он закрепиться или уступит место другим, более упрочившимся стандартам (возможно, микроформатам для несложных применений и какой-либо модифицированной форме RDFa для более сложных). Но в любом случае, время на изучение микроданных не будет потрачено даром. Как мы увидим в следующем разделе, этот подход уже поддерживается Google. Кроме этого, он очень похож на RDF — до такой степени, что в случае, если один из этих форматов возьмет верх над другим, можно будет переключиться на него без больших трудностей.

Чтобы создать блок микроданных, нужно добавить атрибуты `itemscope` и `itemtype` в любой элемент (хотя логичнее будет создать элемент `<div>`, если такого еще нет). Атрибут `itemscope` указывает на начало нового фрагмента семантического содержимого, а атрибут `itemtype` — конкретный тип данных, которые помечаются:

```
<div itemscope itemtype="http://data-vocabulary.org/Person">
  ...
</div>
```

Тип данных обозначается предопределенной, однозначной текстовой строкой, называющейся *пространством имен XML*. В данном примере пространством имен XML является формат для кодирования контактной информации **http://data-vocabulary.org/Person**.

Пространства имен XML часто даются в виде URL. Иногда по этому URL можно даже просмотреть описание соответствующего типа данных, открыв его в браузере. Но пространства имен XML не обязательно должны соотноситься с настоящими веб-сайтами, а также они не обязательно должны быть указаны в виде URL. Название пространства имен формата просто зависит от того, что лицо (или лица) выберет для него при создании этого формата. В этом отношении преимуществом URL является то, что он может содержать доменное имя, принадлежащее лицу или организации. Это повышает шансы названия пространства имен в плане однозначности, т. е. что никто не создаст иной формат данных с таким же названием пространства имен и тем самым собьет всех с толку.

Следующий шаг после создания элемента-контейнера — это использование внутри его атрибута `itemprop`, чтобы отловить важные единицы информации. Для этого применяется такой же базовый подход, что и для микроформатов — используется распознаваемое название `itemprop`, которое позволит другим программным средствам извлекать информацию из связанных элементов. По существу, самая большая разница между микроданными и микроформатами состоит в использовании в первых для маркировки данных атрибута `itemprop`, а не атрибута `class`, как в последних.

```
<div itemscope itemtype="http://data-vocabulary.org/Person">
  <span itemprop="name">Mike Rowe Formatte</span>.
  ...
</div>
```

Микроданные также несколько больше структурированы, чем микроформаты. Например, в них часто один тип данных вкладывается в другой. Таким образом, для подробной контактной информации можно вложить в нее сведения об адресе. Технически, вся информация об адресе относится к отдельному типу данных, который определяется другим пространством имен XML. Поэтому нам нужен новый элемент `<div>` или `<span>` с атрибутами `itemscope` и `itemtype`, как показано в следующем коде:

```
<div itemscope itenrtype="http://data-vocabulary.org/Person">
  ...
  <span itemprop="address" itemscope
      itemtype="http://data-vocabulary.org/Address">
    <span itemprop="street-address">42 Jordan Gordon Street,
        6th Floor</span><br>
    <span itemprop="locality">San Francisco</span>,
    <span itemprop="region">CA</span>
  </span>
</div>
```

Применяя эти правила, исходную разметку страницы "About Me" *(см. разд. "Обозначение контактной информации с помощью микроформата hCard" ранее в этой главе)* можно пометить в формате микроданных. Результаты такого форматирования, в которых внесенные изменения выделены жирным шрифтом, показаны в следующем листинге:

```
<h1>About Me</h1>

<div itemscope itenrtype="http://data-vocabulary.org/Person">
  <img itemprop="photo" src="face.jpg" alt="Mike's Face">
  <p>This website is the work of
  <span itemprop="title" style="display:none">Web Developer</span>
  <b itemprop="name">Mike Rowe Formatte</b>.
  His friends know him as <b itemprop="nickname">The Big M</b>.</p>

  <p>You can contact him where he works, at
  <span itemprop="affillation">The Magic Semantic Company</span> (phone
  <span itemprop="tel">641-545-0234</span> and ask for Mike).</p>

  <p>Or, visit him there at:<br>
  <span itemprop="address" itemscope
       itemtype="http://data-vocabulary.org/Address">
    <span itemprop="street-address">42 Jordan Gordon Street,
                                      6th Floor</span><br>
```

```
    <span itemprop="locality">San Francisco</span>,
    <span itemprop="region">CA</span>
    <span itemprop="postal-code">94105</span><br>
    <span itemprop="country-name">USA</span><br>
  </span>
  <a itemprop="url" href="http://www.magicsemantics.com">
                                    www.magicsemantics.com</a>
</div>
```

Как вы, наверное, заметили, это форматирование поразительно похоже на форматирование посредством стандарта hCard *(см. разд. "Обозначение контактной информации с помощью микроформата hCard" ранее в этой главе)*. Хотя названия свойств сместились с атрибута `class` в атрибут `itemprop`, в этом форматировании все равно используются все те же названия свойств (за исключением названий `fn` и `org`, которые были заменены на `name` и `affiliation` соответственно). А если вы знаете формат RDFa, то обнаружите еще большее соответствие, потому что как в микроданных, так и в RDFa используется один и тот же формат данных **http://data-vocabulury.org/Person**.

Примечание

Все три стандарта для данных с расширенной семантикой — RDFa, микроданные и микроформаты — очень похожи друг на друга. Не то, чтобы эти стандарты были полностью совместимы, но их форматирование достаточно сходное, вследствие чего навыки, приобретенные при изучении одной системы, можно применить для работы с другими.

Микроданные имеют один значительный недостаток: на текущий момент не существует ни подключаемых модулей для браузеров, ни JavaScript-сценариев, которые могли бы извлекать помеченные этим форматом данные из страницы. Но микроданные повышают шансы попадания страницы в результаты поиска, что мы и рассмотрим в следующем разделе.

Расширенные фрагменты страницы

Если вы битком набьете свою веб-страницу семантическими подробностями, то заслужите довольно серьезную репутацию веб-фаната. Но даже самым заядлым веб-разработчикам нужна какая-то отдача, чтобы оправдать требующиеся для этого дополнительные усилия (да и более загроможденную разметку, полученную в результате). Хорошо мечтать о мире сверхинтеллектуальных, понимающих семантику браузерах, но на сегодняшний день жестокая правда жизни такова, что для веб-браузеров существует лишь несколько экспериментальных и малоизвестных модулей расширения, способных извлекать семантически помеченные данные из страницы.

К счастью, способ получить отдачу от использования расширенной семантики в своих веб-страницах все же есть. Называется этот способ *оптимизацией под поисковые системы*. Это искусство сделать веб-сайт более заметным для поисковых

систем. Иными словами, это способ частого попадания веб-сайта в результаты поиска, улучшения его ранжирования по определенным ключевым словам, а также повышение вероятности, что создатель запроса для поиска выберет именно этот веб-сайт из многих других результатов поиска. Для решения последней из этих задач и предназначен способ представления результатов поиска, разработанный компанией Google — *расширенные фрагменты страниц* (rich snippets). Заключается он в том, что поисковая система Google использует семантическую информацию о содержащейся в странице информации, чтобы представить эту информацию в усовершенствованном виде, вследствие чего данный веб-сайт будет выделяться из других результатов поиска.

Но прежде чем приступить к использованию расширенных фрагментов страниц, нужно получить более подробное представление об этом термине. Расширенными фрагментами страниц компания Google называет данные, помеченные семантическими стандартами RDFa, микроформатами или микроданными. Как мы уже узнали из вышеизложенного материала, все эти три подхода очень похожи друг на друга и предназначены для решения одной и той же задачи. Но Google ставит перед собой целью рассматривать их все как равноправные, поэтому не имеет значения, какой из этих подходов вы предпочитаете. (В следующих примерах используются микроданные, чтобы помочь вам лучше разобраться с самым новым семантическим стандартом языка HTML5.)

Для дополнительной информации о стандартах, поддерживаемых расширенными фрагментами страниц, обратитесь к документации Google по адресу **http://tinyurl.com/GoogleRichSnippets**. Здесь можно найти не только приличный обзор RDFa, микроформатов и микроданных, но также много разнообразных примеров расширенных фрагментов (например, контактная информация, события, товары, обзоры, рецепты и т. п.). Для каждого примера есть версия под RDFa, микроформаты и микроданные, что может помочь вам преобразовать свои семантические навыки из одного стандарта в другой, если возникнет такая нужда.

Расширенные результаты поиска

Посмотреть, как работают расширенные фрагменты, можно с помощью инструмента Rich Snippets Testing Tool (RSTT, инструмент тестирования расширенных фрагментов). Этот инструмент исследует предоставленную ему веб-страницу, показывает семантические данные, которые поисковая система Google может извлечь из этой страницы, а потом показывает, каким образом Google может использовать эту информацию, чтобы представить страницу наилучшим образом в результатах чьего-либо поиска.

Примечание

Инструмент RSTT полезен по двум причинам. Первая: он помогает проверять семантическую разметку. (Если Google не может извлечь всю информацию, которую вы поместили на страницу, или если помещает некоторую из этой информации в неправильную категорию, вы знаете, что где-то сделали ошибку.) Вторая: он показывает, каким образом семантические данные могут изменить ее представление в результатах поиска Google.

Порядок использования инструмента RSTT следующий:

1. Откройте в браузере страницу **www.google.com/webmasters/tools/richsnippets**. Эта простая страница содержит одно текстовое поле (рис. 3.6).
2. Введите в текстовое поле URL страницы, семантику которой вы хотите исследовать. К сожалению, инструмент RSTT работает только со страницами, предоставляемыми веб-серверами, поэтому с его помощью нельзя проверить страницу, хранящуюся на локальном жестком диске.

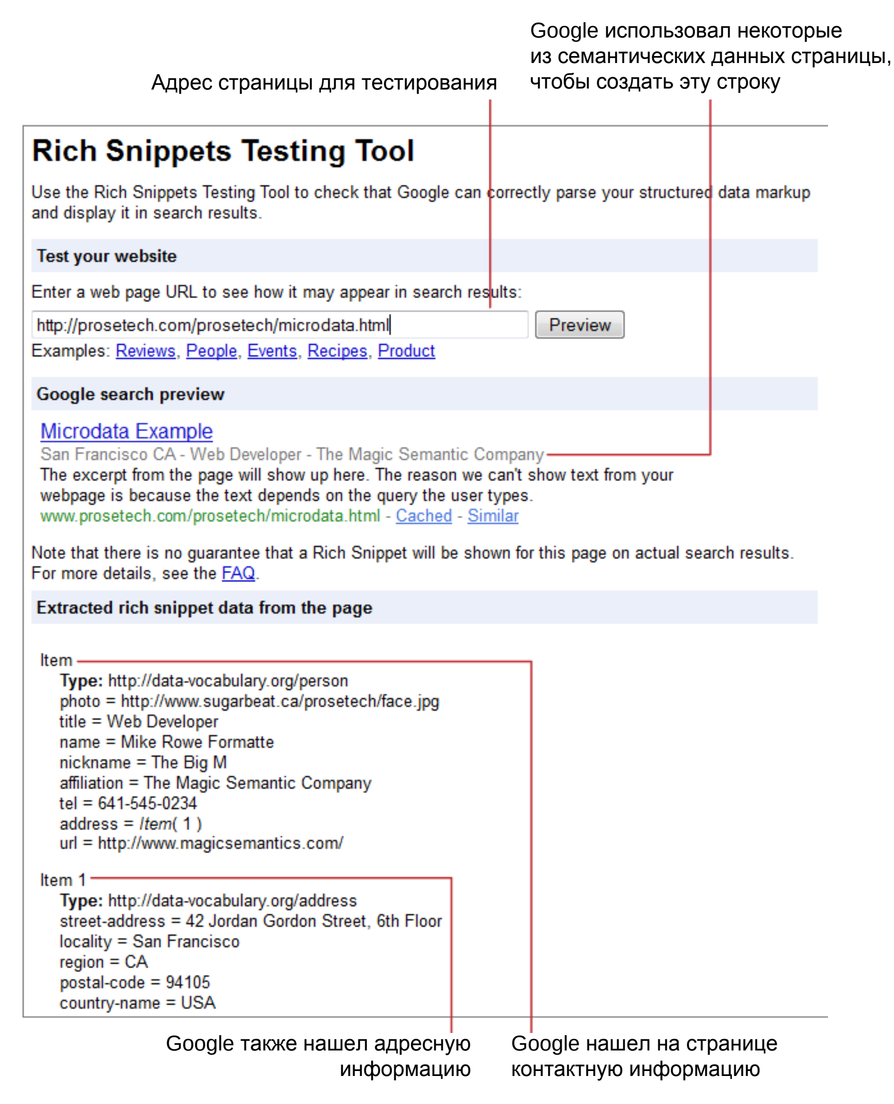


Рис. 3.6. В данном примере Google нашел в странице контактную и адресную информацию лица (см. пример на основе разметки микроданными в *разд. "Микроданные" ранее в этой главе*). Используя эту информацию, Google вставил текстовую строку серого цвета под строкой с заголовком страницы, включив в нее некоторую персональную информацию

3. Нажмите кнопку **Preview**. В окне браузера будут выведены результаты анализа (см. рис. 3.6). В этих результатах особый интерес представляют два раздела. В разделе **Google search preview** показано, как страница может отображаться в результатах поиска. А в разделе **Extracted rich snippet data from the page** отображаются все необработанные семантические данные, которые Google смог извлечь из разметки страницы.

Совет

Если Google выводит сообщение об ошибке "Insufficient data to generate the preview", то возможны три причины. Первая: ваша разметка может быть неправильной. Исследуйте извлеченные Google необработанные данные и убедитесь, что он нашел все помеченные вами данные. Если здесь все в порядке, то, возможно, вы применили тип данных, которые Google еще не поддерживает, или же не предоставили обязательный минимальный набор свойств, требуемый Google. Чтобы определить причину проблемы, сравните свою разметку с одним из примеров Google на сайте **http://tinyurl.com/GoogleRichSnippets**.

Google использует довольно умеренный способ выделения контактной информации. Но контактная информация — это только один из типов расширенных данных, распознаваемых Google. В *разд. "Обозначение событий с помощью микроформата hCalendar" ранее в этой главе* мы рассмотрели, как определять события, используя микроформаты. Такие данные также могут быть включены в результаты поиска Google для страницы (рис. 3.7).

The Fillmore New York at **Irving Plaza** Concert Tickets, Schedule ...
Buy The Fillmore New York at **Irving Plaza** tickets and find concert schedules, venue information, and seating charts for The Fillmore New York at Irving ...
Led Zeppelin 2 — Sat, Jan 23
Cheap Trick with Jason Falkner ... — Mon, Jan 25
Hip Hop Karaoke Championship — Fri, Jan 29
www.livenation.com/.../the-fillmore-new-york-at-**irving-plaza**-new-york-ny-tickets -
Cached - Similar -

Рис. 3.7. В этом примере страницы есть три события. Если для события предоставить URL (как сделано для этого примера), Google оформляет каждое событие как ссылку на соответствующий URL

Google также находит информацию о компаниях (которая обрабатывается по большому счету как и персональная контактная информация), рецепты (которые мы рассмотрим вкратце в следующем разделе) и обзоры (которые рассматриваются далее в этом разделе).

В следующем примере показан текст обзора заведения, отформатированный микроданными. Тип данных для разметки обзора определен на **http://data-vocabulary.org/Review**. Ключевые свойства включают предмет обозрения (в данном случае ресторан), обозревателя и описание. Можно также включить одно предложение общего впечатления (свойство `summary`), дату, когда обозрение было сделано или опубликовано (свойство `dtreviewed`, которое поддерживает элемент `<time>` из HTML5), а также оценку по шкале от 0 до 5 (свойство `rating`).

Разметка с микроданными, выделенными жирным шрифтом, выглядит таким образом:

```
<div itemscope itemtype="http://data-vocabulary.org/Review">
  <h1 itemprop="itemreviewed">Jan's Pizza House</h1>
  <p>Reviewed by <span itemprop="reviewer">Jared Elberadi</span> on
  <time itemprop="dtreviewed" datetime='2011-01-26">January 26</time>.<p>
  <p itemprop="summary">Pretty bad, and then the
                        Health Department showed up.</p>
  <p itemprop="description">I had an urge to mack on some pizza, and this
   place was the only joint around. It looked like a bit of a dive,
   but I went in hoping to find an undiscovered gem. Instead, I watched a
   Health Department inspector closing the place down. Verdict? I didn't
   get to finish my pizza, and the inspector recommends a Hep-C shot.</p>
  <p>Rating: <span itemprop="rating">0.5</span></p>
</div>
```

Страницу с такими данными Google обрабатывает особым образом (рис. 3.8).

Jan's Pizza House
Review by Jared Elberadi - Jan 26, 2011
The excerpt from the page will show up here. The reason we can't show text from your webpage is because the text depends on the query the user types.
www.sugarbeat.ca/prosetech/microdata_review.html - Cached - Similar

Рис. 3.8. Этот обзор по-настоящему выделяется среди других результатов поиска. Звездочки рейтинга бросаются в глаза и немедленно привлекают интерес к этой странице

АВАРИЙНАЯ СИТУАЦИЯ

Что делать, если Google не обращает внимания на ваши семантические данные?

Сам факт, что Google может отображать особым способом семантически расширенную страницу, еще не означает, что он обязательно это сделает для конкретной страницы. Он применяет свой набор полусекретных правил, чтобы определить, представляет ли семантическая информация интерес для создателя запроса. Но если вы не хотите, чтобы Google наверняка игнорировал ваши данные, убедитесь в том, что ваша разметка не удовлетворяет одному из следующих утверждений.

- **Семантические данные не отвечают главному содержимому страницы.** Иными словами, если вы влепите свою контактную информацию на страницу о рыбной ловле, Google, скорее всего, не возьмет вашу контактную информацию. (Ведь когда поисковый движок исследует эту страницу, он, вероятнее всего, ищет что-то имеющее отношение к рыбной ловле, а адрес и название бизнеса не имеют никакого смысла в этом отношении.) С другой стороны, если вставить контактную информацию на страницу резюме, вероятность ее использования будет намного выше.

- **Семантические данные скрыты.** Google не использует НИКАКОГО содержимого, скрытого посредством CSS.

- **Веб-сайт содержит очень небольшой объем семантических данных.** Если веб-сайт содержит мало страниц с семантическими данными, Google может нечаянно пропустить их.

Старайтесь избегать этих ошибок, и вы можете рассчитывать на расширенное отображение вашей страницы в результатах поиска.

Движок для поиска кулинарных рецептов

Расширенное отображение результатов поиска — это, безусловно, ловкий прием, который может направить дополнительный трафик на ваш веб-сайт. Вместе с этим хочется получить что-то еще более впечатляющее за свои усилия, затраченные на овладение навыками работы с семантическими данными. К счастью, гениальные разработчики Google не сидят без дела и воплощают будущее области поиска, и по всем признакам этим будущим окажется семантика.

Одна блестящая идея в этом отношении состоит в использовании семантической информации не для того, чтобы улучшить отображение страницы в результатах поиска, а для того, чтобы повысить избирательность поиска. Например, если соискатели работы будут помечать свои резюме с помощью RDFa, микроформатов или микроданных, Google может предоставить специализированное средство для поиска резюме, которое сканирует эти данные, просматривая резюме на всех популярных работодательных веб-сайтах и игнорируя все другие типы веб-содержимого. Такой движок поиска резюме также можно было бы оснастить улучшенными фильтрационными возможностями, чтобы позволить компаниям находить кандидатов, которые, например, имеют конкретную квалификацию или работали на определенные компании.

В настоящее время Google не имеет такого движка поиска резюме, но недавно эта компания выпустила что-то концептуально очень похожее и, возможно, более практичное: средство для поиска кулинарных рецептов.

К этому времени вы, наверное, уже можете представить себе, как выглядят данные кулинарных рецептов, помеченные микроформатом или микроданными. Весь рецепт заключается в контейнер, который имеет формат данных рецептов (вот этот формат — **http://data-vocabulary.org/Recipe**). Контейнер имеет отдельные свойства для названия рецепта, имени составителя и фотографии готового продукта. Можно также добавить свойство для одного предложения обзора и свойство для пользовательского рейтинга.

В следующем листинге показана часть разметки рецепта:

```
<div itemscope itemtype="http://data-vocabulary.org/Recipe">
  <h1 itemprop="name">Elegant Tomato Soup</h1>
  <img itemprop="photo" src="soup.jpg" alt="A bowl of tomato soup">
  <p>By <span itemprop="author">Michael Chiarello</spanx/p>
  <p itemprop="summary">Roasted tomatoes are the key to
           developing the rich flavor of this tomato soup.</p>
   ...
```

После этого можно добавить ключевую информацию о рецепте, в том числе время подготовки, время приготовления и объем готового блюда. Можно также добавить вложенный раздел с информацией о пищевой ценности (включая размер порции, калорийность, жирность и т. п.):

```
...
<p>Prep time: <time itemprop="prepTime" datetime="PT30M">30 min</time></p>
<p>Cook time: <time itemprop="cookTime" datetime="PT1H">40 min</time></p>
```

```
<p>Yield: <span itemprop="yield">4 servings</span></p>
<div itemprop="nutrition" itemscope
  itemtype="http://data-vocabulary.org/Nutrition">
  Serving size: <span itemprop="servingSize">1 large bowl</span>
  Calories per serving: <span itemprop="calories">250</span>
  Fat per serving: <span itemprop="fat">3g</span>
</div>
...
```

> ***Примечание***
>
> Свойства `prepTime` и `prepTime` представляют не конкретное время, а период времени, поэтому для них не используется тот же формат, что и для элемента HTML5 `<time>`. Вместо этого применяется формат ISO, информацию о котором можно найти по адресу **http://en.wikipedia.org/wiki/ISO_8601#Durations** и **http://ru.wikipedia.org/wiki/ISO_8601**.

Потом следует список ингредиентов. Для каждого ингредиента создается отдельный вложенный блок, который обычно содержит такую информацию, как название и количество ингредиента:

```
...
<ul>
  <li itemprop="ingredient" itemscope
    itemtype="http://data-vocabulary.org/RecipeIngredient">
    <span itemprop="amount">1</span>
    <span itemprop="name">yellow onion</span> (diced)
  </li>
  <li itemprop="ingredient" itemscope
    itemtype="http://data-vocabulary.org/RecipeIngredient">
    <span itemprop="amount">14-ounce can</span>
    <span itemprop="name">diced tomatoes</span>
  </li>
  ...
</ul>
...
```

Выполнение этой части разметки требует значительных усилий, но пусть вас это не останавливает, т. к. награда за эти усилия уже близка.

Наконец, инструкции по приготовлению в нескольких абзацах или списке шагов. Вставлены они в одно свойство:

```
...
<div itemprop="instructions">
  <ol>
    <li>Preheat oven to 450 degrees F.</li>
    <li>Strain the chopped canned tomatoes, reserving the juices.</li>
    ...
  </ol>
  ...
</div>
```

Всю разметку этого рецепта можно просмотреть по адресу:

http://support.google.com/webmasters/bin/answer.py?hl=ru&answer=173379 (или **http://tinyurl.com/RichSnippetsRecipe**).

Примечание

Рецепты обычно довольно длинные и содержат много подробностей, поэтому размечать — их долгая и сложная работа. Это очевидный случай, когда хорошее средство разработки могло бы облегчить работу. В идеале, это средство должно предоставить возможность вводить сведения рецепта в текстовые поля аккуратно упорядоченных окон, а потом генерировать семантически правильную разметку, которую можно будет вставить в страницу.

После того как Google занесет в индекс вашу размеченную страницу с рецептом, этот рецепт можно будет найти с помощью средства Google для просмотра рецеп-

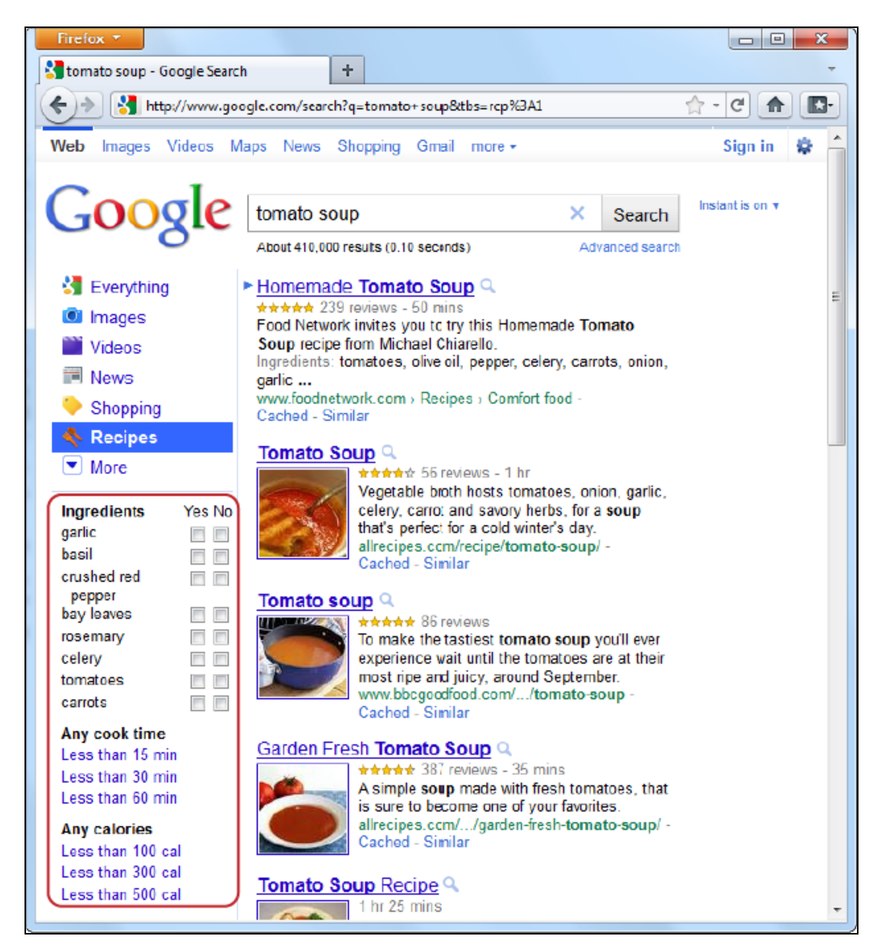


Рис. 3.9. После вывода результатов поиска рецепта Google предоставляет возможность отфильтровать эти результаты на основе информации из расширенных фрагментов, которые он нашел в рецептах, отвечающих запросу. Например, можно указать вхождение или отсутствие определенных ингредиентов, максимальное время приготовления или количество калорий

тов. Для этого откройте в браузере страницу **www.google.com/landing/recipes**, введите название рецепта (можно ввести и на русском языке) в поле поиска, а потом нажмите кнопку **Search** (Поиск). Далее следует интересная часть. Так как Google может понимать структуру каждого рецепта, он в состоянии игнорировать страницы, не содержащие данных настоящих рецептов, а также позволяет указать дополнительные критерии поиска (рис. 3.9).

Из всего этого можно сделать окончательный вывод, что семантические данные предоставляют фанатам Интернета мощный инструмент для поиска информации и более эффективный способ поиска веб-страниц.



ЧАСТЬ II

Создание современных веб-страниц

4

ГЛАВА 4

Продвинутые веб-формы

HTML-формы — это простые элементы управления HTML, которые применяются для сбора информации от посетителей веб-сайта. К ним относятся текстовые поля для ввода данных с клавиатуры, списки для выбора предопределенных данных, флажки для установки параметров и т. п. Существует бесчисленное количество способов использования HTML-форм, и если вы побродили по просторам Интернета всего лишь несколько дней, то, несомненно, использовали их для разных целей — от регистрации на каком-либо форуме или получения почтового ящика до просмотра биржевого курса или покупки товара в интернет-магазине.

HTML-формы существовали с самых ранних времен языка HTML, и с тех пор они нисколько не изменились, несмотря на определенные серьезные усилия. Разработчики веб-стандартов несколько лет колдовали над стандартом XForms, который должен был заменить HTML-формы, но его постиг такой же провал, как и стандарт XHTML 2 *(см. разд. "XHTML 2: неожиданный провал" главы 1)*. Хотя стандарт XForms позволял легко и элегантно решать некоторые задачи, он также имел и значительные недостатки. Например, код XForms был очень объемистый, и для работы с ним нужно хорошее знание стандарта XML. Но самое большое препятствие состояло в том, что стандарт XForms не был совместим с HTML-формами ни в каких отношениях. Это означало, что разработчикам нужно было бы бросаться в неизведанные воды новой модели без вспомогательных плавсредств, а лишь со слепой верой и огромным мужеством. Но так как разработчики основных браузеров никогда не заморачивались с реализацией XForms в своих продуктах по причине его слишком большой сложности и небольшого использования, сообщество веб-разработчиков так никогда и не сделало этот прыжок.

Стандарт HTML5 предлагает другой подход. Вместо того чтобы начинать с нуля, как в XForms, он совершенствует уже существующую модель HTML-форм. Это означает, что HTML5-формы могут работать и на старых браузерах, лишь без новых примочек и наворотов. HTML5-формы также позволяют применять новые возможности, которые уже используются разработчиками в настоящее время. Эти возможности более доступны, не требуют написания страниц сценариев JavaScript или применения инструментариев JavaScript сторонних разработчиков.

В этой главе мы рассмотрим все новые возможности HTML5-форм и увидим, какие из них поддерживаются, а какие — нет, а также взглянем на обходные решения, которые помогут справиться с возможными затруднениями. В завершение мы осудим возможность, которая хотя технически и не является частью стандарта форм HTML5, все-таки демонстрирует все новые интерактивные возможности — внедрение полнофункционального HTML-редактора в обычную веб-страницу.

Что такое форма?

Скорее всего, вам приходилось работать с формами раньше. Но если нет или если вы порядочно подзабыли эту тему, следующий материал позволит вам получить необходимые сведения для более углубленного изучения этой области веб-дизайна.

Веб-форма — это набор текстовых полей, списков, кнопок и других активизируемых щелчком мыши элементов управления, посредством которых посетитель страницы может предоставить ей тот или иной вид информации. Формы в Интернете повсюду — благодаря формам мы можем создавать учетные записи электронной почты, просматривать и покупать товары в интернет-магазинах, осуществлять финансовые транзакции и многое другое. Самая простая форма — это одинокое текстовое поле поисковых систем, таких как Google (рис. 4.1).

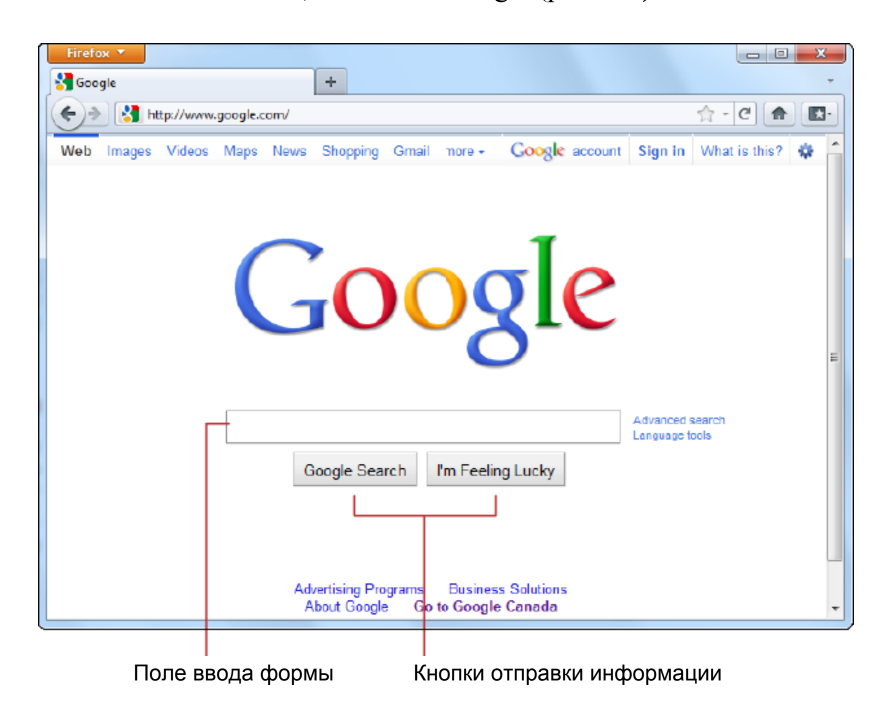


Рис. 4.1. Аскетическая страница системы поиска Google содержит базисную HTML-форму

Все основные веб-формы работают одинаково. Пользователь вводит определенную информацию, а потом нажимает кнопку, чтобы отправить введенную информацию на веб-сервер. По прибытию на веб-сервер эта информация обрабатывается каким-

либо приложением, которое потом предпринимает соответствующий очередной шаг. Перед тем как отослать новую страницу назад браузеру, серверная программа может обратиться к базе данных, чтобы извлечь или сохранить информацию.

Сложность этого процесса состоит в том, что существуют сотни разных способов реализации серверного приложения, которое обрабатывает поступившие из формы данные. Некоторым разработчикам может быть достаточно элементарных сценариев для манипулирования полученными данными, в то время как другие могут использовать средства высшего уровня, которые упаковывают данные из формы в аккуратные программные объекты. Но в любом случае, задача перед этими приложения стоит, по большому счету, одинаковая — исследовать данные из формы, выполнить какие-либо действия с ними, а потом на основе полученных результатов отправить браузеру новую страницу.

Примечание

В этой книге не акцентируется внимание на используемой вами серверной программе. В действительности этот аспект не имеет особого значения, т. к. какую бы серверную программу вы не выбрали, вам все равно нужно будет использовать один и тот же набор элементов управления формы, на которые распространяются одинаковые правила HTML5.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Отправка данных с помощью JavaScript

Стоит заметить, что формы — не единственный способ для отправки введенных пользователем данных на сервер (хотя некогда они и были таковыми). Сегодня хитроумные веб-разработчики могут взаимодействовать с веб-сервером менее заметным образом, используя объект `XMLHttpRequest` *(см. разд. "Объект XMLHttpRequest" главы 11)* в сценариях JavaScript. Например, этот подход применяется в странице поиска Google двумя разными способами. Первый — для предоставления пользователю подсказок в раскрывающемся списке в строке ввода запроса, а второй — чтобы выводить результаты, и все это во время ввода пользователем текста своего запроса. (Но для этого нужно включить возможность Google Instant в настройках поиска этой поисковой системы.)

В связи с этим может возникнуть мысль, что отправку формы нажатием кнопки можно вообще заменить интерактивным обменом данными между формой и сервером, как это делается в Google Instant. Хотя такой способ *можно* было бы предложить в виде опции, он *неприемлем* как единственный и обязательный, потому что подход с использованием JavaScript не совсем надежный. Например, при медленном подключении с ним могут происходить периодические сбои. Кроме этого, все еще есть пользователи, пусть даже и немногочисленные, чьи браузеры не поддерживают JavaScript или в которых эта поддержка отключена.

Наконец, стоит отметить, что вполне допустимо иметь формы, которые никогда не отправляют свои данные. Вам, наверное, встречались формы, выполняющие простые вычисления. Для этих форм не нужна никакая помощь со стороны сервера, т. к. они выполняют вычисления и выводят результат на текущей странице посредством сценариев JavaScript, код которых включен в разметку страницы.

С точки зрения HTML5, что происходит с данными — отправляются ли они для обработки на сервер через кнопку отправки, обрабатываются ли они локально посредством сценариев JavaScript или же передаются на сервер через объект `XMLHttpRequest` — не имеет существенного значения. Во всех случаях форма для них создается на базе элементов управления HTML-форм.

Модернизация традиционной HTML-формы

Лучший способ обучения работе с формами HTML5 — это взять типичную современную форму и усовершенствовать ее. На рис. 4.2 показана форма, на примере которой мы будем обучаться.

Разметка такой формы до предела проста. Если вам раньше приходилось работать с формами, вы не увидите здесь ничего нового. Прежде всего, весь код формы заключается в элемент <form>:

```
<form id="zooKeeperForm" action="processApplication.cgi">
  <p><i>Please complete the form. Mandatory fields are marked
                                with a </i><em>*</em></p>
  ...
```

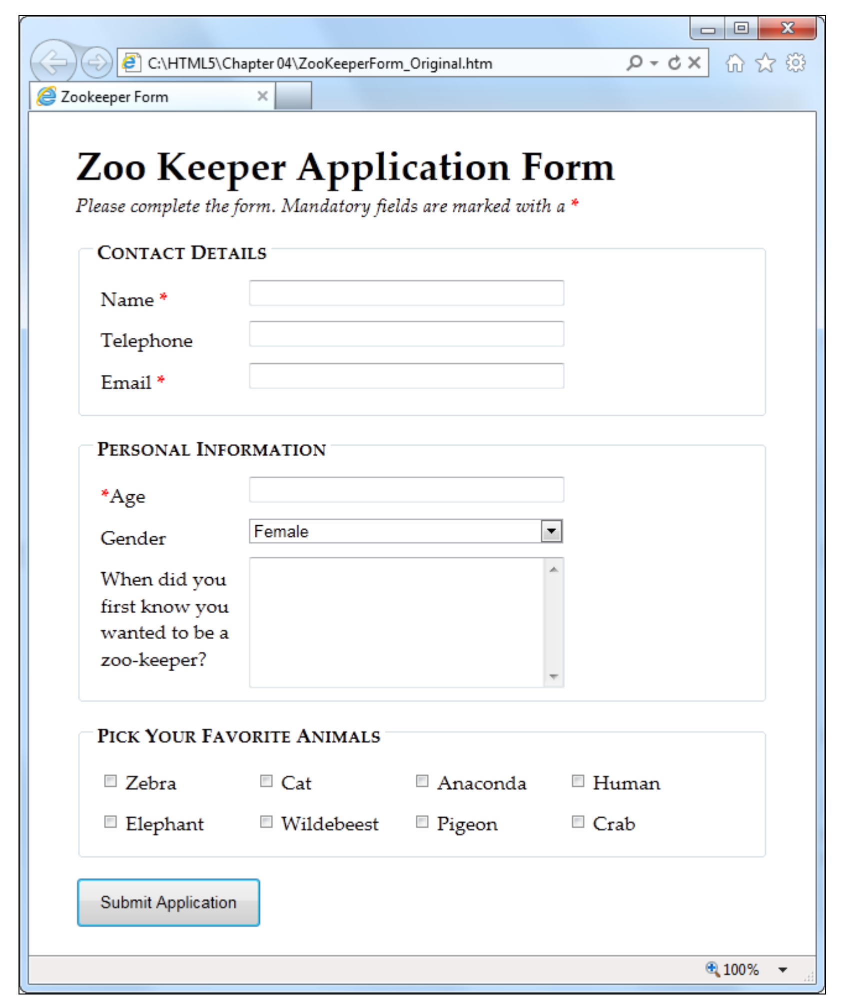


Рис. 4.2. Если вы хоть немного побродили по пространствам Интернета, вам, несомненно, встречалось много форм, подобных этой форме заявления о приеме на работу служителем зоопарка, которые собирают ту или иную информацию от посетителя страницы

Элемент `<form>` удерживает вместе все элементы управления формы, которые также называются *полями*. Кроме этого, он также указывает браузеру, куда отправить данные после нажатия пользователем кнопки отправки, предоставляя URL в атрибуте `action`. Но если вся работа будет выполняться на стороне клиента сценариям JavaScript, то для атрибута `action` можно просто указать значение `#`.

Примечание

В HTML5 добавлен механизм для размещения элементов управления формы вне самой формы. Трюк состоит в использовании нового атрибута `form` для обращения к форме по ее идентификатору (например, `form="zooForm"`). Но браузеры, не поддерживающие эту возможность, совсем не заметят данные такой формы, что делает эту незначительную возможность слишком рискованной для применения в настоящей веб-странице.

Хорошо спроектированная форма, наподобие формы на рис. 4.2, разделяется на логические фрагменты с помощью элемента `<fieldset>`. Каждому разделу можно присвоить название, для чего используется элемент `<legend>`. В следующем листинге приводится разметка раздела формы Contact Details (внешний вид и описание которого показано на рис. 4.3):

```
...
<fieldset>
  <legend>Contact Details</legend>
  <label for="name">Name <em>*</em></label>
  <input id="name"><br>
  <label for="telephone">Telephone</label>
  <input id="telephone"><br>
  <label for="email">Email <em>*</em></label>
  <input id="email"><br>
</fieldset>
...
```

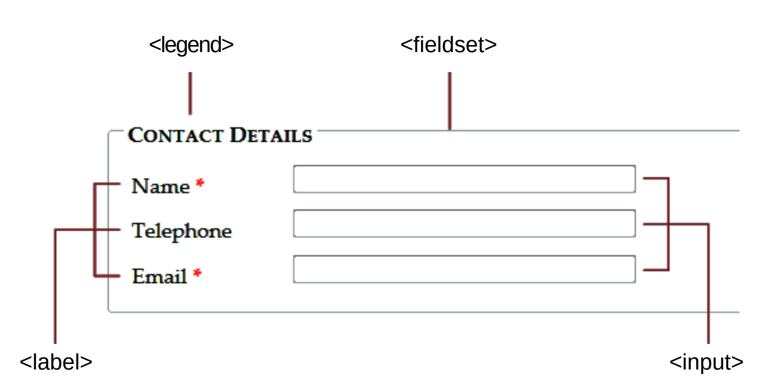


Рис. 4.3. Данный раздел `<fieldset>` собирает три единицы информации: имя, номер телефона и адрес электронной почты. Для каждой единицы информации используются два элемента: элемент `<label>` для обозначения поля ввода данных и элемент `<input>`, `<textarea>` или `<select>` собственно для поля ввода

Как и во всех формах, большая часть работы в нашем примере выполняется универсальным элементом <input>, который собирает данные и создает флажки, переключатели и списки. Для ввода одной строки текста применяется элемент <input>, а для нескольких — элемент <textarea>; элемент <select> создает выпадающий список. Краткое обозрение этих и других элементов управления форм приведено в табл. 4.1.

Таблица 4.1. *Элементы управления формы*

Элемент управления	**HTML-элемент**	**Описание**
Однострочное текстовое поле	`<input type="text">` `<input type="password">`	Выводит однострочное текстовое поле для ввода текста. Если для атрибута `type` указано значение `password`, вводимые пользователем символы отображаются в виде звездочек (*) или маркеров-точек (•)
Многострочное текстовое поле	`<textarea>...</textarea>`	Текстовое поле, в которое можно ввести несколько строчек текста
Флажок	`<input type="checkbox">`	Выводит поле флажка, который можно установить или очистить
Переключатель	`<input type="radio">`	Выводит переключатель — элемент управления в виде небольшого кружка, который можно включить или выключить. Обычно создается группа переключателей с одинаковым значением атрибута `name`, вследствие чего можно выбрать только один из них
Кнопка	`<input type="submit">` `<input type="image">` `<input type="reset">` `<input type="button">`	Выводит стандартную кнопку, активизируемую щелчком мыши. Кнопка типа `submit` всегда собирает информацию с формы и отправляет ее для обработки. Кнопка типа `image` делает то же самое, но позволяет вместо текста на кнопке вставить изображение. Кнопка типа `reset` очищает поля формы от введенных пользователем данных. А кнопка типа `button` сама по себе не делает ничего. Чтобы ее нажатие выполняло какое-либо действие, для нее нужно добавить сценарий JavaScript
Список	`<select>... </select>`	Выводит список, из которого пользователь может выбирать значения. Для каждого значения списка добавляется элемент `<option>`

В следующем листинге приводится оставшаяся часть разметки примера, включающая список <select>, флажки и кнопку для отправки данных:

```
...
<fieldset>
  <legend>Personal Information</legend>
  <label for="age"><em>*</em>Age</label>
  <input id = "age"><br>
  <label for="gender">Gender</label>
  <select id="gender">
    <option value="female">Female</option>
    <option value="male">Male</option>
  </select><br>
  <label for="comments">When did you first know you wanted
                      to be a zoo-keeper?</label>
  <textarea id="comments"></textarea>
</fieldset>

<fieldset>
  <legend>Pick Your Favorite Animals</legend>
  <label for="zebra"><input id="zebra" type="checkbox">Zebra</label>
  <label for="cat"><input id="cat" type="checkbox">Cat</label>
  <label for="anaconda"><input id="anaconda" type="checkbox">
                                             Anaconda </label>
  <label for="human"><input id="human" type="checkbox">Human</label>
  <label for="elephant"><input id="elephant" type="checkbox">
                                             Elephant </label>
  <label for="wildebeest"><input id="wildebeest" type="checkbox">
                                             Wildebeest</label>
  <label for="pigeon"><input id="pigeon" type="checkbox">Pigeon</label>
  <label for="crab"><input id="crab" type="checkbox">Crab</label>
</fieldset>

<p><input type="submit" value="Submit Application"></p>
</form>
```

Полностью разметку этого примера формы, а также относительно простую таблицу стилей для нее, можно загрузить с сайта книги (**www.prosetech.com/html5**). Файл Zookeper-Form_Original.html содержит традиционную версию формы, а файл ZookeperForm_Revised.html — версию HTML5.

Примечание

Одним из ограничений HTML-форм является то, что разработчик не может контролировать, каким способом браузер отображает элементы управления формы. Например, если вы хотите заменить унылое серое поле флажка большим черно-белым полем с жирной красной галочкой, вам этого не удастся. (Одно из решений этой проблемы — создать с помощью JavaScript элемент с поведением, подобным флажку, иными словами, элемент меняет свой внешний вид, когда на нем щелкают.)

Это ограничение сохранилось и в HTML5 и распространяется на все новые элементы управления, которые мы рассмотрим в этой главе. Это означает, что формы не подойдут для разработчиков, которым нужен полный контроль над внешним видом своих страниц в общем и требуются элементы управления с особым внешним видом в частности.

Теперь, когда у нас есть форма, с которой можно работать, настало время улучшить ее с помощью HTML5. Начнем мы это в следующих разделах с добавления подстановочного текста подсказок и поля с автоматическим фокусом.

Добавление подсказок

Обычно поля новой формы не содержат никаких данных. Для некоторых пользователей такая форма может быть не совсем понятной, в частности, какую именно информацию нужно вводить в конкретное поле. Поэтому часто поля формы содержат пример данных, которые нужно ввести в них. Этот *подстановочный текст* также называется "водяным знаком", так он часто отображается шрифтом светло-серого цвета, чтобы отличить его от настоящего, введенного содержимого. Пример такого подстановочного текста показан на рис. 4.4.

Рис. 4.4. *Вверху:* когда поле пустое, в нем отображается подстановочный текст, как в случае с полями **Name** и **Telephone**. *Внизу:* когда пользователь щелкает мышью в поле (устанавливая в нем фокус), подстановочный текст исчезает. Если пользователь переходит в другое поле, не введя ничего в первое, то поле снова заполняется подстановочным текстом

Подстановочный текст для поля создается с помощью атрибута `placeholder`:

```
<label for="name">Name <em>*</em></label>
<input id="name" placeholder="Jane Smith"><br>
<label for="telephone">Telephone</label>
<input id="telephone" placeholder="(xxx) xxx-xxxx"><br>
```

Браузеры, не поддерживающие подстановочный текст, просто не обращают внимания на атрибут `placeholder`; особенно грешит этим Internet Explorer. К счастью, это не такая уж и большая проблема, т. к. подстановочный текст — всего лишь приятная примочка, не обязательная для функционирования формы. Но если такая ситуация лишает вас сна, по адресу **http://tinyurl.com/polyfills** можно найти уйму JavaScript-сценариев, которые без проблем наставят IE на путь истинный.

В настоящее время не существует стандартного, единообразного способа изменить внешний вид подстановочного текста, например, выделить его курсивом или шрифтом определенного цвета. Со временем разработчики браузеров создадут требуемые для этого обработчики. Но пока либо нужно применять специфические для браузера псевдоклассы (а именно `-webkit-input-placeholder` и `-moz-placeholder`), либо смириться с таким порядком вещей. (Псевдоклассы рассматриваются в *разд. "Селекторы псевдоклассов" приложения 1*.)

А вот псевдокласс `focus` обеспечивается лучшей поддержкой, и его можно использовать, чтобы изменять внешний вид текстового поля при получении фокуса. Например, сделать фон поля более темным, чтобы оно выделялось среди остальных, можно следующим образом:

```
input:focus {
  background: #eaeaea;
}
```

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Создание качественного подстановочного текста

Подстановочный текст не обязательно требуется для каждого текстового поля, а только для таких, где пользователь может испытать затруднения с пониманием типа или формата данных. Например, для поля, обозначенного "Полное имя", вряд ли нужны какие-либо объяснения, но вот что нужно вводить в поле с надписью просто "Имя", как в поле **Name** на рис. 4.4, не столь очевидно. А подстановочный текст ясно показывает, что нужно вводить не только имя, но и фамилию.

Иногда подстановочный текст является возможным значением ввода. Например, в качестве подстановочного текста поля запроса на странице поиска рецептов Google (**http://www.google.com/landing/recipes**) употребляется текст "chicken pasta", ясно давая знать, что нужно ввести часть названия рецепта, а, скажем, не название ингредиента или имя повара, придумавшего его.

В других случаях подстановочный текст указывает способ форматирования вводимого значения. Так, подстановочный текст в поле **Telephone** на рис. 4.4 — (xxx) xxx-xxxx — дает понять, что номер телефона должен состоять из трехзначного междугородного кода и трехзначной, а потом четырехзначной последовательностей цифр собственно номера. Этот подстановочный текст не обязательно означает, что нельзя использовать другой формат, но дает подсказку об одном из возможных форматов.

Но подсказка подсказке рознь, и не следует перебирать через край с подстановочным текстом. Прежде всего, не пытайтесь втиснуть в поле его описание или инструкции. Возьмем, например, поле для ввода кода безопасности кредитной карточки. Многие люди не знают, что это такое. Но объяснение "Три цифры справа от поля подписи на обратной стороне карточки" будет не очень хорошим подстановочным текстом. Вместо этого следует рассмотреть другое место для размещения подсказки, например, добавить примечание под полем ввода. Также можно использовать атрибут `title`, чтобы выводить всплывающую подсказку при наведении курсора мыши на поле, как показано в следующем коде:

```
<label for="promoCode">Promotion Code</label>
<input id="promoCode" placeholder="QRB001"
       title="Your promotion code is three letters
       followed by three numbers">
```

Также не используйте специальные символы в подстановочном тексте с целью показать, что это не настоящее значение данного поля. Например, на некоторых веб-сайтах в качестве подстановочного текста для имени используют *[Иван Сидоров]* вместо *Иван Сидоров*, где квадратные скобки должны, по идее, указывать на то, что это только пример. Но пользователи, скорее всего, не поймут этого, и будут вводить свое имя вместе со скобками.

Фокус: правильное начало

Так как форма предназначена для ввода информации, первым делом после ее загрузки пользователи захотят вводить эту информацию. К сожалению, делать этого они не смогут до тех пор, пока не щелкнут мышью по первому полю или выделят его с помощью клавиши <Tab>, установив, таким образом, *фокус* на этом поле.

Пользователю можно помочь в этом, установив фокус на нужном начальном поле автоматически. Это можно сделать с помощью JavaScript, вызывая метод `focus()` требуемого элемента `<input>`. Но этот подход требует лишней строки кода и иногда может вызывать раздражающие неувязки. Например, особо проворные пользователи могут опередить вызов метода `focus()`, щелкнуть в каком-либо другом поле и начать вводить в нем, а когда метод, наконец, вызовется, пользователь грубо выдвинется из выбранного им поля и переместится в поле, выбранное методом. Но если управлять фокусом может браузер, он может быть несколько более смышленым и перемещать фокус только в том случае, если пользователь еще не выбрал другое поле.

На этой идее основан новый HTML5-атрибут `autofocus`, который можно вставить в элемент `<input>` или `<textarea>` (но только в один элемент формы), как показано в следующем примере:

```
<label for="name">Name<em>*</em></label>
<input id="name" placeholder="Dane Smith" autofocus><br>
```

Уровень поддержки браузерами атрибута `autofocus` примерно такой же, как и атрибута `placeholder`, и означает, что практически все браузеры поддерживают его, за исключением Internet Explorer. Но опять же, эта проблема легко решается. Проверить поддержку атрибута `autofocus` конкретным браузером можно с помощью инструмента Modernizr *(см. разд. "Определение возможностей с помощью Modernizr" главы 1)* и, если необходимо, запускать собственный код для автоматического фокуса. Можно также использовать готовый JavaScript-заполнитель для поддержки автоматического фокуса (**http://tinyurl.com/polyfills**). Но вряд ли стоит прилагать эти усилия ради такой незначительной примочки, если только вы не планируете заодно обеспечить поддержку в IE других возможностей форм, наподобие системы проверки данных, рассматриваемой в следующем разделе.

Проверка: ошибкам — нет!

Поля в форме предназначены для сбора информации от посетителей страницы. Но несмотря на все усилия и объяснения, получение правильной информации может оказаться трудным делом. Нетерпеливые или невнимательные посетители могут

пропускать важные поля, заполнять поле или несколько полей не полностью, да и попросту нажимать неправильные клавиши. Заполнив таким образом форму, они нажимают кнопку **Отправить**, и серверная программа получает набор данных, с которыми она не знает, что делать.

Серьезной веб-странице требуется проверка данных, т. е. какой-либо способ обнаружения ошибок во введенных данных, а еще лучше — способ, не допускающий ошибок вообще. На протяжении многих лет веб-разработчики делали это с помощью процедур JavaScript собственной разработки или посредством профессиональных библиотек JavaScript. И говоря по правде, эти подходы давали отличные результаты. Но, видя, что проверка выполняется повсеместно (практически всем нужно проверять вводимые данные на ошибки) и применяется лишь к нескольким основным типам данных (например, адресам электронной почты или датам), а также, что ее реализация — такая неинтересная и скучная задача (никому, по сути, не хочется писать один и тот же код для каждой формы, не говоря уже о тестировании его), определенно должен быть лучший способ для ее реализации.

Создатели HTML5 смогли увидеть эту надобность и изобрели способ привлечь браузеры на помощь, переложив задачу проверки с плеч веб-разработчиков на веб-обозреватели. Создатели HTML5 разработали систему проверки на *стороне клиента (см. врезку. "На профессиональном уровне. Двусторонняя проверка" далее в этой главе)*, которая позволяет вставлять основные правила валидации в любое поле `<input>`. Лучшее в этой системе — это ее простота и легкость: все, что нужно делать, — так это вставить правильный атрибут.

Как работает проверка HTML5?

Основная идея в основе проверки форм HTML5 состоит в том, что разработчик указывает данные для валидации, но *не реализует* все необходимые для этого трудоемкие подробности. Это что-то похожее на начальника, который только отдает приказания, но реализует эти указания не сам, а с помощью подчиненных.

Например, допустим, что определенное поле нельзя оставлять пустым, и посетитель должен ввести в него хоть что-то. В HTML5 это осуществляется с помощью атрибута `required` в соответствующем поле:

```
<label for="name">Name <em>*</em></label>
<input id="name" placeholder="Jane Smith" autofocus required><br>
```

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Двусторонняя проверка

На протяжении нескольких лет изобретательные веб-разработчики подходили к решению задачи проверки с разных сторон, но на сегодняшний день достигнуто общее мнение. Чтобы полностью исключить возможность ввода в форму неправильных данных, требуются два вида проверки на ошибки.

- **Проверка на стороне клиента.** Эта проверка выполняется браузером до отправки формы на сервер. Ею цель — облегчить пользователю заполнение полей формы. Вместо того чтобы дать ему заполнить три с половиной десятка полей, отправить форму для обработки, получить ответ от сервера о некорректности данных и пред-

ложить заполнить форму снова, ошибки ввода следует улавливать в процессе их возникновения. Таким образом, можно сразу же вывести полезное сообщение об ошибке в соответствующем месте, позволив пользователю исправить ее, прежде чем отправлять форму на сервер.

- **Проверка на стороне сервера.** Эти проверки формы выполняются после ее отправки на сервер на стороне сервера. На данном этапе серверное приложение проверяет полученные данные, чтобы удостовериться в их соответствии требованиям, прежде чем приступать к их дальнейшей обработке. Независимо от проверки на стороне браузера, проверка на стороне сервера является обязательной. Это единственный способ обезопасить себя от злоумышленников, которые намеренно пытаются предоставить неправильные данные. Если серверная программа проверки обнаружит проблему с данными, она отправит в браузер страницу с сообщением об ошибке.

Таким образом, проверка на стороне клиента (примером которой является проверка с помощью HTML5) предназначена для облегчения заполнения формы посетителями веб-страницы, а проверка на стороне сервера призвана обеспечить правильность введенных данных. Ключевой момент, который нужно понимать, — это обязательное выполнение проверки обоих типов, если только речь не идет о чрезвычайно простой форме, допущение ошибок на которой маловероятно или корректность данных не играет большой роли.

Даже с применением для поля атрибута `required` на это требование нет никаких визуальных указателей по умолчанию. Поэтому следует обратить внимание пользователя на это требование с помощью каких-либо своих визуальных признаков, например, выделив рамку поля цветом и поставив возле него звездочку (как на рис. 4.2).

Проверка выполняется, когда пользователь нажмет кнопку для отправки формы. Если браузер поддерживает формы HTML5, он заметит пустое обязательное для заполнения поле, перехватит вызов отправки формы и отобразит всплывающее сообщение об ошибке (рис. 4.5).

Как мы увидим в последующих разделах, с помощью других атрибутов можно выполнять проверку на ошибки ввода других типов. К одному полю можно применять несколько правил, а одно и то же правило — к скольким угодно элементам `<input>`, как и к элементу `<textarea>`. Прежде чем браузер даст разрешение на отправку введенных данных, все требования валидации должны быть удовлетворены.

В связи с этим возникает вопрос: что будет, если нарушено несколько правил проверки, например, не заполнено несколько обязательных полей?

Опять же, ничего не будет, пока пользователь не нажмет кнопку для отправки формы. Только после этого браузер начнет проверять поля сверху вниз. Встретив первое некорректное значение, он прекращает дальнейшую проверку, отменяет отправку формы и выводит сообщение об ошибке рядом с полем, вызвавшим эту ошибку. (Кроме этого, если при заполнении формы область с полем ошибки вышла за пределы экрана, браузер прокручивает экран, чтобы это поле находилось вверху страницы.) После того как пользователь исправит данную ошибку и опять нажмет кнопку для отправки формы, браузер остановится на следующей ошибке ввода, и процесс повторится.

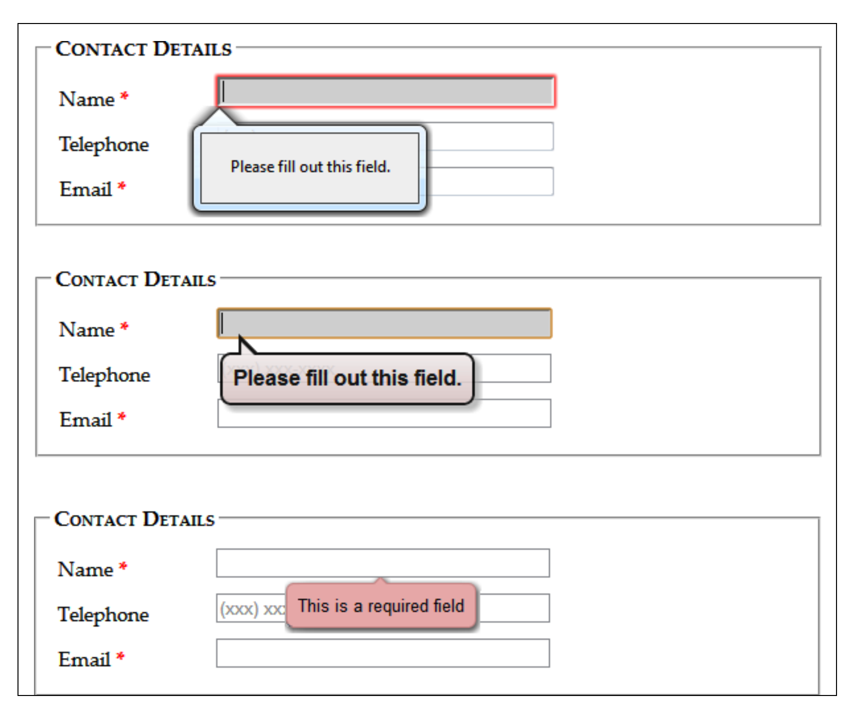


Рис. 4.5. Сообщение о пустом обязательном для заполнения поле в браузерах Firefox *(вверху)*, Chrome *(в центре)* и Opera *(внизу)*. Хотя официальных требований к оформлению сообщения об ошибках при проверке не существует, во всех браузерах для этой цели применяются всплывающие подсказки. К сожалению, веб-разработчики не могут изменять оформление или текст этого сообщения, по крайней мере не в настоящее время

Примечание

Браузеры выполняют проверку введенных пользователем данных только после того, как тот нажмет кнопку для их отправки. Таким образом обеспечивается эффективность и умеренность системы проверки, чтобы она работала для всех.

Некоторым веб-разработчикам хотелось бы предупреждать пользователя об ошибке сразу же после выхода из неправильно заполненного поля (когда он нажмет клавишу <Tab> или щелкнет мышью в другом поле). Такой подход к проверке полезен для длинных форм, особенно если возможно, что пользователь сделает одну и ту же ошибку в нескольких полях. К сожалению, HTML5 не предоставляет разработчикам средства указывать браузеру, когда ему выполнять проверку, хотя она может быть добавлена в будущем. Тем временем, возможность немедленного оповещения об ошибках ввода лучше всего реализовать с помощью своего JavaScript-сценария или же воспользоваться хорошей библиотекой JavaScript.

Отключение проверки

В некоторых случаях может потребоваться отключить проверку. Например, вы хотите протестировать свой серверный код на правильность обработки поступивших некорректных данных. Чтобы отключить проверку для всей формы, в элемент `<form>` добавляется атрибут `novalidate`:

```
<form id="zooKeeperForm" action="processApplication.cgi" novalidate>
```

Другой подход — это отключить проверку в кнопке для отправки формы. Такой способ иногда полезен в настоящей веб-странице. Например, вместо отправки, может быть, требуется сохранить наполовину заполненную форму для дальнейшего использования. Чтобы получить такую возможность, в элемент `<input>` соответствующей кнопки вставляется атрибут `formnovalidate`:

```
<input type="submit" value="Save for Later" formnovalidate>
```

Обнаружение незаполненных полей — это только один из нескольких типов проверки. Далее мы рассмотрим обнаружение ошибок в данных разных типов.

> **ПРИМЕЧАНИЕ**
>
> Планируете выполнять проверку чисел? В настоящее время не существует правила проверки наличия цифр в тексте, но есть новый тип данных `number`, который мы рассмотрим в *разд. "Числа" далее в этой главе*. К сожалению, поддержка этого типа данных все еще поверхностная.

Оформление результатов проверки

Хотя веб-разработчики не могут оформлять сообщения об ошибках проверки, они могут изменять внешний вид полей в зависимости от *результатов* их валидации. Например, можно выделить поле с неправильным значением цветным фоном сразу же, когда браузер обнаружит неправильные данные.

Все, что для этого требуется, — это добавить несколько простых псевдоклассов *(см. разд. "Селекторы псевдоклассов" приложения 1)*. Доступны следующие опции:

- `required` и `optional`, которые применяют форматирование к полю в зависимости от того, использует ли это поле атрибут `required` или нет;
- `valid` и `invalid`, которые применяют форматирование к полю в зависимости от правильности введенного в него значения. Но не забывайте, что большинство браузеров не проверяет данные, пока пользователь не попытается отправить форму, поэтому форматирование полей с некорректными значениями не выполняется сразу же при введении такого значения;
- `in-range` и `out-of-range`, применяющие форматирование к полям, для которых используется атрибут `min` или `max`, чтобы ограничить их значение определенным диапазоном значений *(см. разд. "Числа" далее в этой главе)*.

Например, сделать желтым фон полей `<input>`, обязательных для заполнения, можно с помощью правила CSS с псевдоклассом `required`:

```
input:required {
  background-color: lightyellow;
}
```

А сделать желтым фон только тех полей, которые являются обязательными и в настоящее время содержат некорректные значения, можно с помощью комбинации псевдоклассов `required` и `invalid` таким образом:

```
input:required:invalid {
  background-color: lightyellow;
}
```

При такой настройке незаполненные поля автоматически выделяются желтым цветом, т. к. они нарушают правило `required` для поля.

Можно применять всяческие другие приемы, например, использование псевдоклассов для проверки совместно с псевдоклассом `focus` или пометку некорректных значений посредством смещенного фона со значком ошибки. Но в то время как псевдоклассы могут улучшить внешний вид веб-страниц, использовать их следует с некоторой мерой предосторожности. В частности, надо удостовериться, что страница выглядит должным образом без форматирования псевдоклассами, т. к. уровень их поддержки некоторыми браузерами оставляет желать лучшего.

Проверка с помощью регулярных выражений

Самым мощным (и самым сложным) поддерживаемым HTML5 типом проверки является проверка на основе регулярных выражений. Поскольку JavaScript уже поддерживает регулярные выражения, добавление этой возможности к формам HTML будет вполне логичным шагом.

Регулярное выражение — это шаблон для сопоставления с образцом, закодированный согласно определенным синтаксическим правилам. Регулярные выражения применяются для поиска в тексте строк, которые отвечают определенному шаблону. Например, с помощью регулярного выражения можно проверить, что почтовый индекс содержит правильное число цифр, или в адресе электронной почты присутствует знак `@`, а его доменное расширение содержит, по крайней мере, два символа. Возьмем, например, следующее выражение:

```
[A-Z]{3}-[0-9]{3}
```

Квадратные скобки в начале строки определяют диапазон допустимых символов. Иными словами, группа `[A-Z]` разрешает любые прописные буквы от A до Z. Следующая за ней часть в фигурных скобках указывает множитель, т. е. `{3}` означает, что нужны три прописные буквы. Следующее тире не имеет никакого специального значения и означает самое себя, т. е. указывает, что после трех прописных букв должно быть тире. Наконец, `[0-9]` обозначает цифры в диапазоне от 0 до 9, а `{3}` требует три таких цифры.

Регулярные выражения полезны для поиска — нахождения в тексте строк, отвечающих условиям, заданных в выражении, и проверки, что определенная строка отвечает заданному регулярным выражением шаблону. В формах HTML5 регулярные выражения применяются для валидации.

Примечание

Внимание: для обозначения начала и конца значения в поле символы `^` и `$`, соответственно, не требуются. HTML5 автоматически предполагает наличие этих двух символов. Это означает, что значение в поле должно *полностью* совпасть с регулярным выражением, чтобы его можно было считать корректным.

Таким образом, следующие значения будут допустимыми для регулярного выражения `[A-Z]{3}-[0-9]{3}`, рассмотренного ранее:

```
QRB-001
TTT-952
LAA-000
```

А вот эти нет:

```
qrb-001
TTT-0952
LA5-000
```

Но регулярные выражения очень быстро становятся более сложными, чем рассмотренный нами пример. Поэтому создание правильного регулярного выражения может быть довольно трудоемкой задачей, что объясняет, почему большинство разработчиков предпочитает использовать для проверки данных на своих страницах уже готовые регулярные выражения. Или же они обращаются за профессиональной помощью.

> ***Совет***
>
> Получить достаточно информации о регулярных выражениях для создания сверхпростых выражений можно с помощью кратких руководств по адресу **www.w3schools.com/js/js_obj_regexp.asp** или **http://tinyurl.com/jsregex**. Готовые регулярные выражения для своих форм можно найти на сайте **http://regexlib.com**. А чтобы стать настоящим докой в регулярных выражениях, прочитайте книгу Джеффри Фридла "Регулярные выражения"[1].

Чтобы применить полученное тем или иным путем регулярное выражение для проверки значения поля `<input>` или `<textarea>`, его следует добавить в этот элемент в качестве значения атрибута `pattern`:

```
<label for="promoCode">Promotion Code</label>
<input id="promoCode" placeholder="QRB-001" title=
 "Your promotion code is three uppercase letters, a dash,
                                            then three numbers"
  pattern="[A-Z]{3}-[0-9]{3}">
```

На рис. 4.6 показана реакция браузера, если введенное значение не проходит проверку.

> ***Совет***
>
> Браузеры не поддерживают проверку пустых значений. В примере на рис. 4.6 пустое значение пройдет проверку. Если вам нужно нечто иное, можно вдобавок к атрибуту `pattern` использовать атрибут `required`.

> ***Примечание***
>
> Регулярные выражения кажутся идеальным средством для проверки правильности адресов электронной почты. Впрочем, не кажутся, а так оно и есть. Но подождите не-

[1] Фридл Дж. Регулярные выражения. — СПб.: Символ-Плюс, 2008.

много с использованием их таким образом, т. к. в HTML5 уже имеется выделенный тип для адресов электронной почты со встроенным в него регулярным выражением *(см. разд. "Адреса электронной почты" далее в этой главе)*.

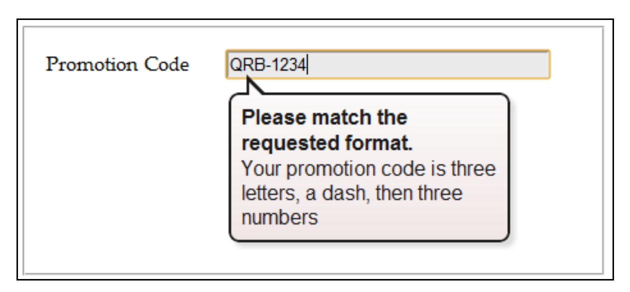


Рис. 4.6. Умные браузеры (такие, как Chrome, использованный для этого примера) не только улавливают ошибку, но также выводят в сообщении об ошибке текст атрибута `title`, чтобы помочь пользователю, заполняющему форму

Специализированная проверка

Спецификация HTML5 также оговаривает набор свойств JavaScript, с помощью которых можно определить корректность значений полей (или заставить браузер выполнить проверку). Наиболее полезным из них является метод `setCustomValidity()`, с использованием которого можно написать специальный сценарий для проверки конкретных полей, который будет работать с системой валидации HTML5.

Осуществляется это следующим образом. Прежде всего, значение соответствующего поля проверяется на правильность. Это делается с помощью обычного события `oninput`:

```
<label for="comments">When did you first know you wanted to
                                                be a zookeeper?</label>
<textarea id="comments" oninput="validateComments(this)"></textarea>
```

В данном примере событие `oninput` активизирует функцию `validateComments()`. Ответственность за написание этой функции, проверку текущего значения поля `<input>` и вызова метода `setCustomValidity()` лежит на разработчике.

Если с текущим значением поля имеются проблемы, при вызове метода `setCustomValidity()` ему необходимо передать сообщение об ошибке. Если же текущее значение допустимо, этот метод вызывается с пустой строкой, таким образом очищая специальные сообщения об ошибке, которые могли использоваться ранее.

Далее приводится пример использования этого метода для проверки требования, чтобы значение в поле комментариев содержало не менее 20 символов:

```
function validateComments(input) {
  if (input.value.length < 20) {
    input.setCustomValidity("You need to comment in more detail.");
  }
```

```
  else {
    // Длина комментария отвечает требованию.
    // Очищаем предыдущие сообщения об ошибке.
    input.setCustomValidity("");
  }
}
```

На рис. 4.7 показан результат срабатывания этой функции при попытке отправить форму со слишком коротким комментарием.

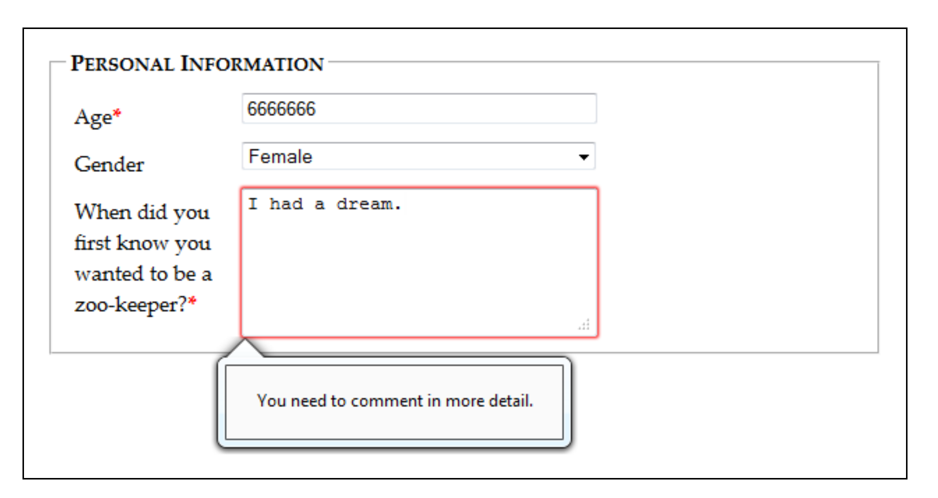


Рис. 4.7. Когда методу `setCustomValidity()` передается сообщение об ошибке, браузер обрабатывает его как собственное, встроенное сообщение об ошибке при проверке

Конечно, эту задачу можно решить более элегантно с помощью регулярного выражения, требующего длинную строку. Но в то время как регулярные выражения прекрасно подходят для проверки определенных типов данных, специальная логика проверки может делать *все, что угодно*, от сложных алгебраических вычислений до установления связи с веб-сервером.

Примечание

Имейте в виду, что посетители сайта могут видеть весь JavaScript-код, поэтому он не подходит для реализации секретных алгоритмов. Например, для проверки правильности кода товара, продаваемого по акции, нужно удостовериться, что сумма его цифр равна 12. Эту проверку вряд ли стоит реализовывать в специальной процедуре валидации на JavaScript, т. к. могут найтись ловкачи, которые разберутся с вашим сценарием, создадут фальшивые акционные коды, и тогда вы не напасетесь акционного товара. Такую проверку следует осуществлять на стороне веб-сервера.

Поддержка проверки браузерами

Разработчики браузеров добавляли поддержку проверки в свои продукты по частям, вследствие чего некоторые версии браузеров поддерживают одни возможности валидации, но не обращают внимания на другие. В табл. 4.2 указаны мини-

мальные версии основных браузеров, которые предоставляют солидную поддержку всем методам проверки, рассмотренным на данном этапе.

Таблица 4.2. *Поддержка проверки браузерами*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
Минимальная версия	10*	4	10	5 (только для Windows)	10	—	—

* На момент написания этой книги доступны только ранние бета-сборки этой версии.

Так как проверка HTML5 не заменяет валидацию на стороне сервера, ее можно рассматривать как второстепенную возможность, когда даже такая несовершенная поддержка лучше, чем отсутствие вообще какой-либо поддержки. В браузерах, не поддерживающих проверку, таких как IE 9, можно отправлять формы с некорректными данными, но эти ошибки можно выявить на стороне сервера и возвратить эту страницу назад браузеру, но с указанными ошибками.

С другой стороны, ваш веб-сайт может содержать сложные формы, в которых можно сделать массу ошибок при вводе данных, и вы не хотите потерять тех IE-пользователей, которые после первой неудачной попытки заполнить вашу форму не предпримут другую. В таком случае у вас есть два пути: разработать и использовать свою систему проверки или же использовать библиотеку JavaScript, чтобы компенсировать умственную отсталость IE. Какой из этих двух подходов выбрать, зависит от объема и сложности проверки.

Если нужно выполнить простую проверку небольшого количества полей, скорее всего, стоить сделать это своими силами. С помощью инструмента Modernizr *(см. разд. "Определение возможностей с помощью Modernizr" главы 1)* можно проверить поддержку браузером разнообразных возможностей HTML5-форм. Например, выяснить, поддерживает ли браузер атрибут `pattern` можно с помощью свойства `Modernizr.input.pattern` таким образом:

```
if (!Modernizr.input.pattern)  {
  // Данный браузер не поддерживает проверку регулярными выражениями.
  // Выполняйте проверку посредством JavaScript.
  ...
}
```

ПРИМЕЧАНИЕ

Свойство `pattern` — всего лишь одно из свойств, предоставляемых объектом `Modernizr.input`. Другие свойства, которые могут быть полезными для проверки поддержки браузерами форм, — это `placeholder`, `autofocus`, `required`, `max`, `min` и `step`.

Конечно же, в этом примере не показано, *когда* выполнять эту проверку и *каким образом* реагировать на ее результаты. Если нужно, чтобы ваша процедура проверки походила на систему валидации HTML5, имеет смысл выполнять проверку,

когда пользователь пытается отправить форму. Для этого нужно обрабатывать событие формы `onSubmit`, после чего возвращать значение `true` (означающее, что данные корректны и браузер может их отправить) или `false` (означающее, что с данными имеются проблемы и браузер должен отменить отправку):

```
<form id="zooKeeperForm" action="processApplication.cgi"
      onsubmit="return validateForm()">
```

В следующем листинге приведен очень простой пример процедуры специальной проверки, которая обеспечивает правильность заполнения требуемых полей:

```
function validateForm() {
  if (!Modernizr.input.required) {
    // Браузер не поддерживает требуемый атрибут;
    // проверяем необходимые поля своими силами.
    // Первым делом, получаем массив, содержащий все элементы.
    var inputElements = document.getElementById("zooKeeperForm").elements;

    // Далее сканируем все элементы массива.
    for(var i = 0;  i < inputElements.length;  i++)  {
      // Проверяем, требуется ли этот элемент.
      if (inputElements[i].hasAttribute("required"))  {
        // Если данный элемент требуется, проверяем, содержит ли
        // он значение.
        // Если не содержит, форма не проходит проверку,
        // и эта функция возвращает значение false.
        if (inputElements[i].value == "") return false;
      }
    }

    // Если мы достигли этой точки, все данные корректны,
    // и браузер может отправлять их.
    return true;
  }
}
```

СОВЕТ

В данном фрагменте кода используется несколько основных методов программирования JavaScript, включая нахождение элементов, цикл и условные переходы. Более подробную информацию *см. в приложении 2*.

Если же вы хотите сэкономить время и усилия на проверке значений полей сложной формы (и в то же самое время подготовиться к будущему), вам, может быть, лучше использовать заплатку JavaScript для решения всех своих задач. Технически, подход такой же, как и в первом случае — страница проверяет поддержку браузером возможностей валидации и сама выполняет требуемые проверки, которые не поддерживаются браузером. Разница заключается в том, что библиотека JavaScript уже содержит весь требуемый код.

По адресу **http://tinyurl.com/polyfills** можно найти длинный список библиотек JavaScript, которые все, по большому счету, делают то же самое. Одна из лучших среди этих библиотек — это *webforms2*; загрузить ее можно по адресу **https://github.com/westonruter/webforms2**.

Библиотека *webforms2* реализует все рассмотренные на данный момент атрибуты. Для использования библиотеки загрузите все ее файлы в папку своего веб-сайта (а лучше в подкаталог папки веб-сайта) и добавьте в веб-странице следующую ссылку:

```
<head>
  <title>...</title>
  <script src="webforms2.js"></script>
  ...
<head>
```

Библиотека *webforms2* хорошо интегрируется с другой заплаткой JavaScript, называющейся *html5Widgets*. Она реализует поддержку возможностей форм, которые мы рассмотрим далее, таких как ползунок и средства выбора даты и цвета. Более подробную информацию об этой универсальной заплатке см. на веб-сайте **www.useragentman.com/tests/html5Widgets**. Обе эти библиотеки предоставляют хорошую общую поддержку для веб-форм, но содержат в своем коде неизбежные пробелы и незначительные ошибки. Качество сопровождения и усовершенствования этих библиотек покажет только время.

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ
Несколько необычных атрибутов элемента `<input>`

Стандарт HTML5 признает еще несколько атрибутов, используемых для управления браузером при заполнении форм, но не для проверки. Не все эти атрибуты поддерживаются всеми браузерами.

Тем не менее, с ними хорошо экспериментировать.

- **Атрибут `spellcheck`.** Некоторые браузеры пытаются заполнить пробелы в знаниях правописания пользователя, проверяя орфографию вводимого текста. Очевидная проблема с этой услугой заключается в том, что не весь текст состоит из настоящих слов, и роспись волнистых красных подчеркиваний может очень быстро начать действовать пользователю на нервы. Чтобы браузер не проверял орфографию текста в поле, присвойте атрибуту `spellcheck` значение `false`, а для проверки — значение `true`. (По умолчанию разные браузеры действуют по-разному в отношении проверки орфографии, а установка атрибута `spellcheck` приводит к единообразному поведению.)
- **Атрибут `autocomplete`.** Некоторые браузеры пытаются сэкономить время пользователя, предлагая при вводе информации в поле значения, которые вводились в это поле ранее. Такое поведение не всегда желательно — как указывается в спецификации HTML5, некоторая информация может быть конфиденциальной (например, коды для запуска ядерных ракет) или оставаться актуальной только непродолжительное время (например, одноразовый пароль входа в банковскую систему самообслуживания). Для таких полей установите значение атрибута `autocomplete` в `off`, чтобы браузер не предлагал возможных вариантов завершения вводимого в поле текста. А чтобы выполнять автозаполнение для определенного поля, установите значение его атрибута `autocomplete` в `on`.

- **Атрибуты `autocorrect` и `autocapitalize`.** Эти атрибуты применяются для управления возможностями автоматического исправления и капитализации на некоторых мобильных браузерах, а именно в версиях Safari для iPad и iPhone.
- **Атрибут `multiple`.** Веб-разработчики использовали атрибут `multiple` с элементом `<select>` для создания списков с множественным выбором с незапамятных времен. Но сейчас они могут использовать этот атрибут с определенными типами элемента `<input>`, включая тип `file` (для закачивания файлов) и `email` *(см. разд. "Адреса электронной почты" далее в этой главе)*. В браузере, поддерживающем этот атрибут, пользователь может выбрать сразу несколько файлов для закачивания на сервер или вставить несколько адресов электронной почты в одно поле.

Новые типы элемента *<input>*

Одной из странных особенностей HTML-форм является использование одного элемента с расплывчатым названием `<input>` для создания разнообразных элементов управления, от флажков до текстовых полей и кнопок. Конкретный вид элемента управления зависит от атрибута `type` элемента `<input>`.

Если браузеру встречается неизвестный тип элемента `<input>`, веб-обозреватель рассматривает его как обычное текстовое поле. Это означает, что следующие три элемента обрабатываются абсолютно одинаково всеми браузерами:

```
<input type="text">
<input type="странный-престранный-тип-input">
<input>
```

В HTML5 из этого поведения извлекается польза. А именно, в элемент `<input>` было добавлено несколько новых типов, и если какой-либо браузер не поддерживает их, он будет обрабатывать их как обычные текстовые поля. Например, для ввода адреса электронной почты можно создать поле `<input>` нового типа `email`:

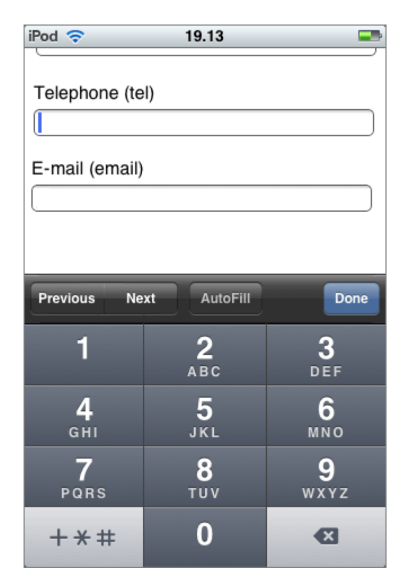
```
<label for="email">Email <em>*</em></label>
<input id = "email" type="email"><br>
```

Если просматривать страницу с этим кодом в браузере, который не поддерживает тип `email` для элемента `<input>` (например, Internet Explorer), то это поле отобразится как обычное текстовое поле. Но браузеры, поддерживающие формы HTML5, немного умнее и могут делать следующее.

- **Предложить услуги редактирования.** Например, интеллектуальный браузер может предоставить способ выбрать адрес электронной почты из адресной книги и вставить его в поле адреса.
- **Предотвратить возможные ошибки.** Например, браузер может не принимать буквы при вводе в поле типа `number`, или не принимать недопустимые даты, или вообще заставить пользователя выбирать даты из мини-календаря, что легче и безопаснее.
- **Выполнять проверку.** Браузеры могут выполнять более сложные проверки после того, как пользователь нажмет кнопку для отправки данных. Например, ин-

теллектуальный браузер выявит очевидные ошибки в адресе электронной почты и откажется отправлять данные.

Спецификация HTML5 не предоставляет разработчикам браузеров никаких наставлений по первому пункту. Браузеры свободны управлять отображением и редактированием разных типов данных любым имеющим смысл способом, и разные браузеры могут добавлять различные небольшие удобства. Например, мобильные браузеры могут воспользоваться этой информацией, чтобы специализировать виртуальную клавиатуру под определенное приложение (рис. 4.8).



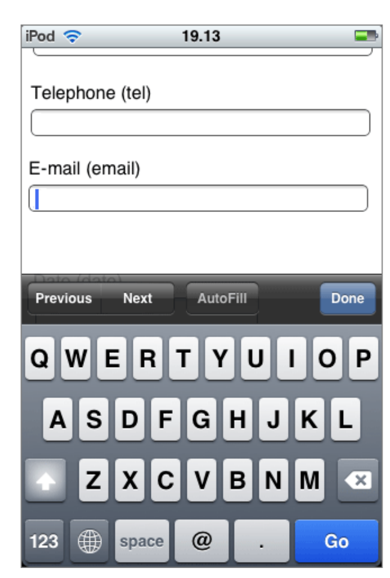


Рис. 4.8. При заполнении формы на мобильном устройстве у пользователей нет возможности работать с полной клавиатурой. В iPod их участь облегчается предоставлением клавиатуры, специализированной под конкретный тип данных. Таким образом, для ввода телефонных номеров предоставляется цифровая клавиатура в стиле номеронабирателя телефона *(слева)*, а для ввода адресов электронной почты — клавиша <@> и уменьшенная клавиша пробела *(справа)*

Но более важными являются возможности проверки и исключения ошибок. Как абсолютный минимум, браузер с поддержкой HTML5-форм должен не допустить отправки формы, содержащей данные, которые нарушают правила типов данных. Поэтому, если браузер не может предотвратить ошибки непосредственно при вводе (согласно второму пункту вышеприведенного списка), он должен выполнить их проверку, когда пользователь попытается отправить форму (согласно третьему пункту).

К сожалению, не все современные браузеры удовлетворяют этим требованиям. Некоторые распознают новые типы данных и предоставляют кое-какие возможности редактирования, но не проверки. Многие браузеры понимают один тип данных, но не другой. Особенно проблемные в этом отношении мобильные браузеры — они предоставляют некоторые удобства редактирования, но никаких возможностей проверки.

В табл. 4.3 приведены новые типы данных и уровень полной поддержки их основными браузерами. Полная поддержка означает, что в случае нарушения правил типа данных форма не отправляется.

Таблица 4.3. *Поддержка новых типов данных основными браузерами*

Тип данных	IE	Firefox	Chrome	Safari	Opera	Safari iOS**	Android
`email`	—	4	10	5 (только для Windows)	10.6	—	—
`url`	—	4	10	5 (только для Windows)	10.6	—	—
`search`*	Нет данных	Нет данных	Нет данных	Нет данных	Нет данных	Нет данных	Нет данных
`tel`*	Нет данных	Нет данных	Нет данных	Нет данных	Нет данных	Нет данных	Нет данных
`number`	—	—	10	5 (только для Windows)	—	—	—
`range`	—	—	6	5	11	—	—
`datetime`, `date`, `month`, `week`, `time`	—	—	10	—	11	—	—
`color`	—	—	—	—	11	—	—

* Стандарт HTML5 не требует проверки для этого типа данных.

** Хотя браузер для iOS не поддерживает проверку, предоставление этим браузером специализированных клавиатур (см. рис. 4.8) является значительным удобством, поэтому в приложениях для этого веб-обозревателя стоит использовать специальные типы данных.

Совет

Кстати, поддержку типов данных можно выполнить с помощью свойств объекта `Modernizr.inputtypes` инструмента Modernizr. Например, если браузер поддерживает тип данных `range`, свойство `Modernizr.inputtypes.range` возвращает значение `true`.

Адреса электронной почты

Тип данных `email` используется для полей, предназначенных для ввода адресов электронной почты. В общем, адрес электронной почты состоит из строки символов (использование некоторых символов не допускается). Допустимый адрес должен содержать символ @ и точку, между которыми должен быть минимум один символ, а после точки — минимум два символа. Вроде бы простые правила, но их

логическая реализация или создание регулярного выражения для проверки адресов электронной почты оказывается на удивление сложной задачей, о которую споткнулся не один решительно настроенный веб-разработчик. Поэтому радует существование браузеров, которые поддерживают тип данных `email` и выполняют проверку этих данных автоматически (рис. 4.9).

Рис. 4.9. Браузер Firefox отказывается принять адрес электронной почты, содержащий пробел

Тип `email` поддерживает атрибут `multiple`, который позволяет вводить несколько адресов в поле. Но эти несколько адресов все равно выглядят, как одна строка текста, только разделены запятыми.

ПРИМЕЧАНИЕ

Не забывайте, что пустые значения проходят проверку. Чтобы не допустить пустого поля адреса, в его элемент нужно вставить атрибут `required` *(см. разд. "Как работает проверка HTML5" ранее в этой главе)*.

URL-адреса

Тип `URL` применяется для полей ввода URL-адресов. Вопрос, что является допустимым URL, продолжает горячо обсуждаться. Но большинство браузеров применяет сравнительно нестрогий алгоритм проверки. Адрес должен содержать префикс (который может быть как настоящим, типа *http://*, так и выдуманным, типа *bonk//*) и позволяет вводить пробелы и большинство специальных символов, за исключением двоеточия (:).

Некоторые браузеры также предлагают возможные варианты URL в выпадающем списке, которые обычно взяты из журнала недавно посещенных браузером страниц.

Поля поиска

Тип `search` применяется для полей поиска. Они обычно предназначены для ввода ключевых слов, по которым потом выполняется какой-либо вид поиска. Это может быть поиск по всему Интернету (как в Google на рис. 4.1), поиск по одной странице или же специальная поисковая процедура, которая исследует каталог информации. В любом случае поле поиска выглядит и ведет себя почти точно так же, как и обычное текстовое поле.

В некоторых браузерах, например Safari, поле поиска выглядит слегка по-другому и имеет скругленные углы. Кроме этого, когда пользователь начинает вводить данные в поле поиска в браузере Safari или Chrome, с правой стороны поля выводится небольшой значок в виде X, щелкнув по которому можно очистить поле. За исключением этих незначительных различий, поле поиска является ничем иным, как обычным текстовым полем. Основная разница заключается в семантике. Иными словами, тип данных `search` используется для того, чтобы сделать ясным назначение поля для браузеров и вспомогательных программ для пользователей с ограниченными возможностями. Они могут направлять посетителей в требуемое место страницы или предоставлять другие, более интеллектуальные услуги — возможно, в будущем.

Телефонные номера

Тип данных `tel` применяется с целью обозначения полей для ввода телефонных номеров, которые могут быть представлены в самых разных форматах. В одних случаях используются только цифры, в других применяются пробелы, тире, знак "плюс" и круглые скобки. Возможно, это отсутствие единого формата и есть причина того, что стандарт HTML5 не требует от браузеров выполнения проверки телефонных номеров. Вместе с тем, не понятно, почему поле типа `tel` не отклоняет, по крайней мере, буквы.

В настоящее время единственная польза от применения поля типа `tel` состоит в предоставлении специализированной виртуальной клавиатуры для ввода телефонных номеров на мобильных браузерах, которая содержит цифры, но не имеет букв.

Числа

В HTML5 определяются два числовых типа данных. Тип `number` предназначен для обычных чисел.

Этот тип данных имеет очевидный потенциал. Обычные текстовые поля принимают буквально все: цифры, буквы, пробелы, знаки пунктуации и знаки, обычно применяемые для обозначения выражения отрицательных эмоций персонажами комиксов. По этой причине одна из самых распространенных задач проверки — убедиться, что значение является числом в определенном диапазоне. Но при вводе данных в поле типа `number` браузер автоматически игнорирует все символы, кроме цифр. Далее показан пример кода для создания поля этого типа:

```
<label for="age">Age<em>*</em></label>
<input id="age" type="number"><br>
```

Конечно же, есть много чисел, которые не подходят для каждого типа числовых данных. Например, в приведенной выше разметке разрешается возраст наподобие 43 000 или –6 лет, что несколько не соответствует реальности. Эта проблема решается с помощью атрибутов `min` и `max`. В следующем коде представлен пример ограничения возраста разумным диапазоном от 0 до 120 лет:

```
<input id="age" type="number" min="0" max="120"><br>
```

Обычно поля типа number принимают только целые числа, а дроби, например 30,5, не разрешаются. (Более того, некоторые браузеры даже не позволят ввести десятичный знак.) Но это поведение также можно изменить с помощью атрибута step, который указывает шаг изменения числа (в большую или меньшую сторону). Например, установив значение step в 0.1, можно вводить такие значения, как 0,1, 0,2 0,3 и т. д. Но попробуйте отправить форму со значением 0,15, и вы получите знакомое всплывающее сообщение об ошибке. По умолчанию значение шага равно 1.

```
<label for="weight">Weight (in pounds)</label>
<input id="weight" type="number" min="50" max="1000" step="0.1"
      value="160"><br>
```

Атрибут step также влияет на работу кнопок поля со счетчиком (рис. 4.10).

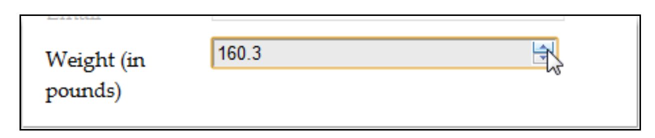


Рис. 4.10. Многие браузеры оформляют поле типа number в виде наборного счетчика. Каждый щелчок по кнопке набора увеличивает или уменьшает значение на величину шага, пока не будет достигнуто максимальное или минимальное установленное значение

Ползунки

Другим числовым типом HTML5 является range. Подобно типу number, этот тип может представлять целые и дробные значения. Также поддерживает атрибуты min и max для установки диапазона значений. Далее показан пример кода для создания поля этого типа:

```
<label for="weight">Weight (in pounds)</label>
<input id="weight" type="range" min="50" max="1000" value="160"><br>
```

Разница состоит в том, каким образом поле типа range представляет свою информацию. Вместо счетчика, как для поля типа number, интеллектуальные браузеры отображают ползунок (рис. 4.11).

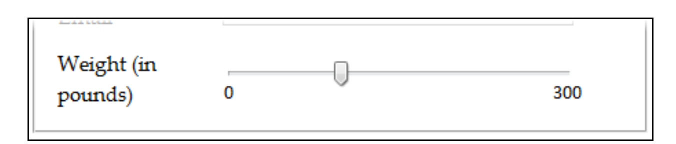


Рис. 4.11. Вам, скорее всего, приходилось использовать элементы управления наподобие поля range для выбора значения в диапазоне, например уровня звука. Поля этого типа хорошо подходят для ввода информации в узком диапазоне с известным минимальным и максимальным значениями, когда важно не точное значение, а его расположение относительно минимума и максимума

Чтобы установить значение типа range, нужно просто перетянуть ползунок в требуемую позицию между минимальным и максимальным значениями. Но браузеры, поддерживающие этот тип поля, не предоставляют никакой обратной информации

об установленном значении. Чтобы получить эти сведения, в разметку нужно добавить процедуру JavaScript, которая реагирует на изменения положения ползунка (возможно, посредством обработки события `onChange`), а потом отображает эту информацию рядом с полем. Конечно же, также нужно проверить (с помощью инструмента наподобие Modernizr), поддерживает ли браузер тип `range`, и если нет, то надобность в исполнении этого кода отпадет, т. к. значение будет отображаться в обычном текстовом поле.

Дата и время

В HTML5 определяется несколько типов данных, связанных со временем. Браузеры, которые поддерживают типы дат, могут выводить удобный выпадающий календарь, в котором пользователь может выбрать требуемую дату и/или время. Это не только устраняет неопределенность относительно правильного формата даты, но также запрещает случайно (или нарочно) установить несуществующую дату. Интеллектуальные браузеры могут делать еще больше, например поддерживать интеграцию с персональным календарем.

В настоящее время, несмотря на их очевидную пользу, поддержка временны́х типов данных находится на низком уровне. Opera — единственный браузер, который предоставляет выпадающий календарь (рис. 4.12). Браузер Chrome обеспечивает минимальную поддержку дат: даты отображаются подобно числам в наборных счетчиках и выполняется проверка введенных значений.

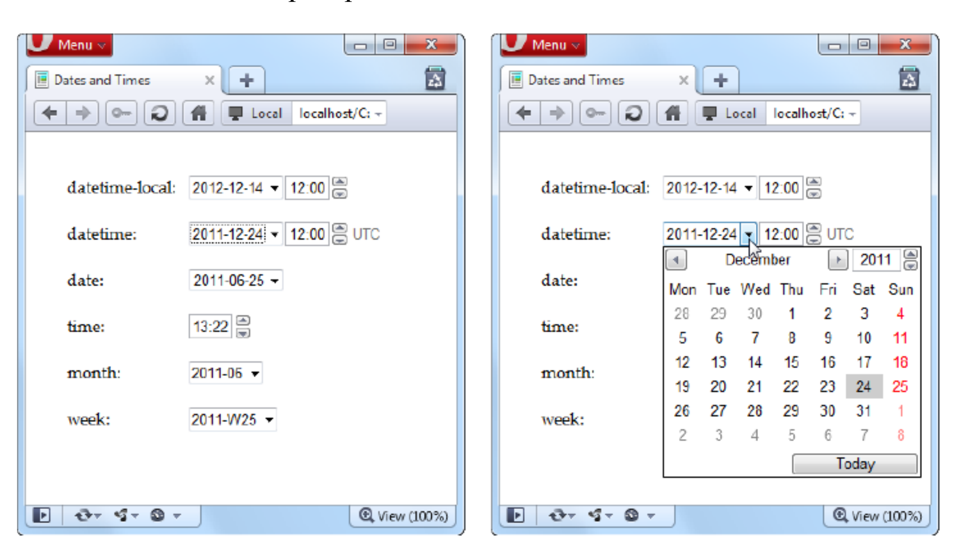


Рис. 4.12. Поле `<input>` отображается несколько по-другому для дат и времени *(слева)*. Но настоящее удобство (к сожалению, предоставляемое только браузером Opera) заключается в выпадающем календаре, с помощью которого можно установить требуемую дату, не беспокоясь, в каком формате это делать

Совет

При использовании этих типов данных подумайте о применении заполнителей (polyfills) для браузеров, которые не поддерживают эти типы, например библиотеку

> *html5Widgets* (**www.useragentman.com/tests/html5Widgets**), рассмотренную в *разд. "Поддержка проверки браузерами" ранее в этой главе*. Идея эта хороша потому, что пользователи с такими браузерами могут легко ошибиться и ввести дату в неправильном формате, а выполнять проверку данных самому и предоставлять необходимые инструкции — слишком трудоемкая задача. (Это объясняет, почему уже существуют элементы управления JavaScript для дат и времени и применяются повсюду в веб-страницах.)

В табл. 4.4 перечислены шесть новых форматов для дат и времени, дано их краткое описание и пример использования.

Таблица 4.4. *Типы данных для дат и времени*

Тип данных	**Описание**	**Пример**
`date`	Дата по шаблону ГГГГ-ММ-ДД	25 января 2012 г.: *2012-01-25*
`time`	Время в 24-часовом формате с необязательными секундами, по шаблону чч:мм:сс.сс	14:35 (и 50,2 секунды): *14:35* или *14:35:50.2*
`datetime-local`	Дата и время, разделенные прописной английской буквой T (по шаблону ГГГГ-ММ-ДД*T*чч:мм:сс)	25 января 2012 г., 14:35: *2012-01-25T14:35*
`datetime`	Дата и время, как и для типа `datetime-local`, но со смещением временного пояса. Используется такой же шаблон (ГГГГ-ММ-ДД*T*чч:мм:сс-чч: мм), как и для элемента `<time>`, рассмотренного в *разд. "Обозначение дат и времени с помощью элемента <time>" главы 3*	25 января 2012 г., 14:35 в Нью-Йорке: *2012-01-25T14:35-05:00*
`month`	Год и номер месяца по шаблону ГГГГ-ММ	Первый месяц 2012 г.: *2012-01*
`week`	Год и номер недели по шаблону ГГГГ-*W*нн. Обратите внимание, что в зависимости от года может быть 52 или 53 недели	Вторая неделя 2012 г.: *2012-W02*

> ***Совет***
>
> Браузеры, которые поддерживают типы данных для дат и времени, также поддерживают атрибуты `min` и `max` для них, что позволяет устанавливать минимальные и максимальные даты при условии использования правильного формата даты. Таким образом, ограничить даты 2012 годом можно следующим кодом: `<input type="date" min="2012-01-01" max="2012-12-31">`.

Цвет

Тип данных `color` применяется для полей, предназначенных для ввода цвета. Тип данных `color` — это интересная, хотя редко используемая второстепенная возможность, позволяющая посетителю веб-страницы выбирать цвет из выпадающей палитры, похожей на палитру графического редактора MS Paint. В настоящее время эту возможность поддерживает только браузер Opera. Для других браузеров нужно

вводить шестнадцатеричный код цвета самому (или использовать библиотеку *html5Widgets* (**www.useragentman.com/tests/html5Widgets**).

Новые элементы

На данном этапе мы узнали, как усовершенствовать формы с помощью новых возможностей HTML5-проверки и дополнительных типов данных полей ввода. Это наиболее важные новые возможности с наивысшим уровнем поддержки в браузерах, но HTML5 не останавливается на этом.

В стандарте также вводится несколько совершенно новых элементов. С их помощью в форму можно добавить список предложений, индикатор выполнения задания, панель инструментов и многое другое. Проблема с этими новыми элементами заключается в том, что старые браузеры точно не поддерживают их, и даже новые браузеры не особо спешат с предоставлением поддержки, пока спецификация окончательно не утверждена. Таким образом, в последующем материале рассматриваются возможности с самым низким уровнем поддержки изо всех возможностей, рассмотренных в этой главе. Желательно разобраться, как они работают, но при этом разумно воздержаться от их использования, пока вы не будете уверены в том, что браузеры их поддерживают.

Подсказки ввода <*datalist*>

Элемент `<datalist>` предоставляет способ присоединить выпадающий список возможных вариантов ввода к обыкновенному текстовому полю. Заполняющим форму пользователям он дает возможность либо выбрать вариант ввода из списка значений, либо ввести требуемое значение вручную (рис. 4.13).

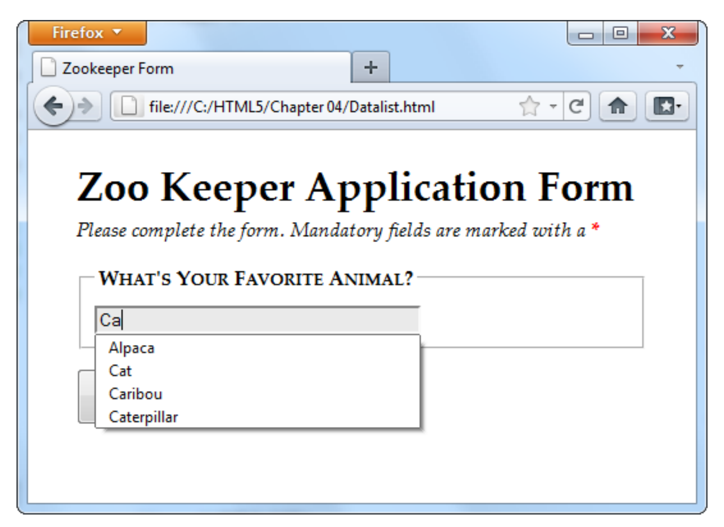


Рис. 4.13. По мере ввода значения в поле браузер предлагает возможное требуемое значение. Например, при вводе букв "са" браузер показывает все названия зверей, которые содержат эту последовательность в любом месте (не обязательно в начале)

Чтобы использовать элемент datalist, сначала нужно создать обычное текстовое поле. Допустим, мы создали обычный элемент <input>:

```
<legend>What's Your Favorite Animal?</legend>
<input id="favoriteAnimal">
```

Чтобы добавить к этому полю выпадающий список возможных значений, для него нужно создать элемент <datalist>. Технически этот элемент можно разместить в любом месте разметки, т. к. он не отображается браузером, а просто предоставляет данные для использования в текстовом поле ввода. Но логично поместить этот элемент либо непосредственно перед тем элементом <input>, для которого он предоставляет свои данные, либо сразу же после него. Далее показан пример кода для создания списка <datalist>:

```
<datalist id="animalChoices">
  <option label="Alpaca"      value="alpaca">
  <option label="Zebra"       value="zebra">
  <option label="Cat"         value="cat">
  <option label="Caribou"     value="caribou">
  <option label="Caterpillar" value="caterpillar">
  <option label="Anaconda"    value="anaconda">
  <option label="Human"       value="human">
  <option label="Elephant"    value="elephant">
  <option label="Wildebeest"  value="wildebeest">
  <option label="Pigeon"      value="pigeon">
  <option label="Crab"        value="crab">
</datalist>
```

Как и традиционное поле <select>, список <datalist> использует элементы <option>. Каждый элемент <option> представляет собой отдельное возможное значение, которое браузер может предложить заполняющему форму. Значение атрибута label содержит текст, который отображается в текстовом поле, а атрибут value — текст, который будет отправлен на сервер, если пользователь выберет данную опцию. Сам по себе список <datalist> не отображается в браузере. Для того чтобы подключить его к текстовому полю, нужно установить значение атрибута list равным значению параметра id соответствующего списка <datalist>:

```
<input id="favoriteAnimal" list="animalChoices">
```

В браузерах, которые поддерживают <datalist> (а в настоящее время это только Opera 10 и Firefox 4 или более поздние версии), посетители увидят результат, как на рис. 4.13. Другие браузеры будут игнорировать атрибут list и разметку <datalist>, делая все предложения возможного ввода бесполезными.

Но эту проблему можно исправить с помощью хитрого резервного решения, которое заставляет старые браузеры вести себя как следует. Трюк заключается в том, чтобы вставить другое содержимое в список <datalist>. Этот подход работает потому, что браузеры, которые поддерживают элемент <datalist>, обращают внимание только на элементы <option> и игнорируют все другое содержимое. В следую-

щем листинге приводится разметка, в которой используется это поведение. (Жирным шрифтом выделен код, который будет игнорироваться браузерами, поддерживающими элемент <datalist>.)

```
<legend>What's Your Favorite Animal?</legend>
<datalist id="animalChoices">
  <span class="Label">Pick an option:</span>
  <select id="favoriteAnimalPreset">
    <option label="Alpaca"      value="alpaca">
    <option label="Zebra"       value="zebra">
    <option label="Cat"         value="cat">
    <option label="Caribou"     value="caribou">
    <option label="Caterpillar" value="caterpillar">
    <option label="Anaconda"    value="anaconda">
    <option label="Human"       value="human">
    <option label="Elephant"    value="elephant">
    <option label="Wildebeest"  value="wildebeest">
    <option label="Pigeon"      value="pigeon">
    <option label="Crab"        value="crab">
  </select>
  <br>
  <span class="Label">Or type it in:</span>
</datalist>
<input list="animalChoices" name="list">
```

Удалив код, выделенный жирным шрифтом, получим практически такую же разметку, как в предыдущем листинге. Это означает, что браузеры, которые распознают элемент <datalist>, будут отображать только одно текстовое поле и выпадающий список предлагаемых вариантов ввода, как показано на рис. 4.13. Но для других браузеров этот дополнительный код облекает предлагаемые варианты ввода в традиционный список <select>, предоставляя пользователю возможность ввести собственное значение или же выбрать готовое значение из списка (рис. 4.14).

Но этот подход не работает незаметно. При получении данных формы на сервере необходимо проверить на наличие данных из списка <select> (favoriteAnimalPreset) и из списка <datalist> (animalChoices). Но за исключением этого незначительного недостатка, это надежный способ предоставить новое удобство всем пользователям без исключения.

Примечание

Элемент <datalist> вначале имел возможность, позволяющую ему получать возможные варианты ввода из каких-либо других источников, например, он мог отправить вызов на веб-сервер, который потом извлек бы список предлагаемых вариантов ввода из базы данных. Эта возможность, может быть, еще будет добавлена в одной из будущих версий стандарта HTML, но в настоящее время реализовать ее можно только с помощью сценария JavaScript, использующего объект XMLHttpRequest *(см. разд. "Объект XMLHttpRequest" главы 11)* для получения данных.



Рис. 4.14. Возможные варианты ввода можно предлагать и в браузерах, которые не поддерживают элемент <datalist>, обернув их сначала в список <select>

Индикатор выполнения *<progress>* и счетчик *<meter>*

Новые графические элементы <progress> и <meter> (рис. 4.15) внешне похожи друг на друга, но имеют разные назначения.

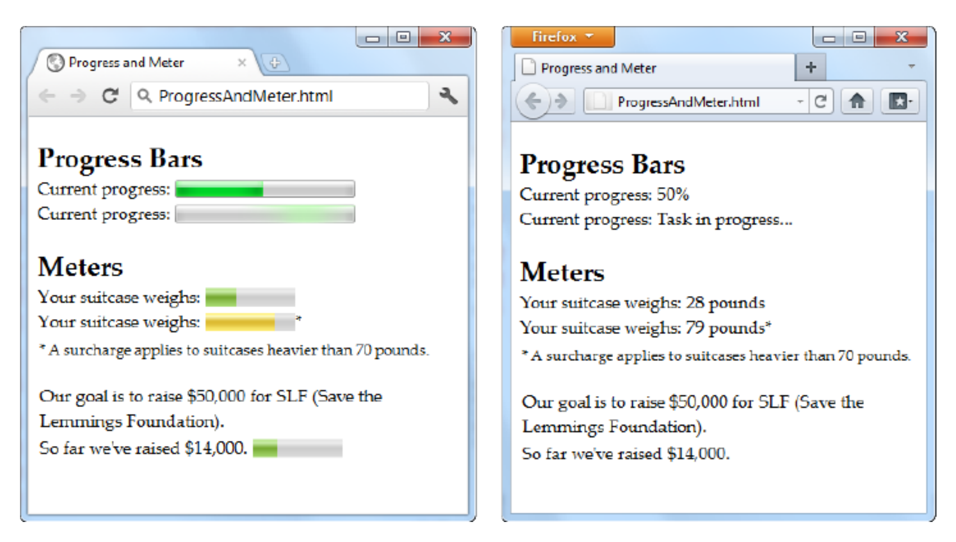


Рис. 4.15. В браузерах, которые поддерживают элементы <progress> и <meter>, эти элементы визуально отображают информацию *(слева)*. В других браузерах эта информация отображается посредством резервного решения *(справа)*

Элемент <progress> отображает ход выполнения задания посредством зеленой пульсирующей полоски на сером фоне. Визуально элемент <progress> похож на индикаторы выполнения, с которыми вам, вероятно, приходилось не раз встречать-

ся (например, индикатор выполнения копирования в операционной системе Windows), хотя его точный внешний вид зависит от браузера, в котором просматривается страница.

Элемент `<meter>` указывает значение в диапазоне известных значений. Внешне он похож на элемент `<progress>`, но зеленая полоска имеет другой оттенок и не пульсирует. В зависимости от браузера цвет индикаторной полоски элемента `<meter>` может меняться при отображении значения, определенного как "слишком низкое" или "слишком высокое". Например, в браузере Chrome при отображении последнего значения цвет индикатора меняется с зеленого на желтый. Но самая важная разница между элементами `<progress>` и `<meter>` заключается в их семантическом значении.

Примечание

Для помещения в форму элементов `<meter>` и `<progress>` нет надобности. Более того, они даже не являются настоящими элементами управления формы, т. к. не собирают информацию от посетителя веб-страницы. Но официальный стандарт HTML5 включает их потому, что в некоторых отношениях они *чувствуются* как элементы формы, возможно, из-за графического отображения данных.

В настоящее время элементы `<progress>` и `<meter>` поддерживаются браузерами Chrome 9, Opera 11 и Safari 5.1 (а также их более поздними версиями). Браузеры Firefox и Internet Explorer эти элементы еще не поддерживают.

Применение элементов `<progress>` и `<meter>` не составляет никакого труда. Сначала рассмотрим элемент `<progress>`. Он использует атрибут `value`, который обозначает ход выполнения задания в виде дробной величины от 0 до 1. Графически это отображается соответствующей шириной полоски индикатора. Например, чтобы показать, что задание выполнено на 25%, атрибуту `value` присваивается значение 0,25:

```
<progress value="0.25"></progress>
```

Альтернативно, можно использовать атрибут `max`, чтобы установить максимальное значение и изменить масштаб индикатора. Например, при значении `max`, равном 200, значение `value` должно быть между 0 и 200. Если сделать значение `value` равным 50, то получим те же самые 25% заполнения индикатора, как и в предыдущем примере:

```
<progress value="50" max="200"></progress>
```

Масштаб используется просто в целях удобства, т. к. цифровое значение в индикаторе не отображается.

Примечание

Элемент `<progress>` — всего лишь средство для отображения нарядного индикатора выполнения. Сам по себе этот элемент ничего не делает. Например, если индикатор выполнения применяется для отображения хода исполняющегося в фоне задания (скажем, используя фоновые вычисления, как рассматривается в *разд. "Выполнение вычислений в фоновом режиме" главы 12*), ответственность за создание JavaScript-кода для изменения значения `value` элемента `<progress>` лежит на веб-разработчике.

Браузеры, которые не поддерживают элемент `<progress>`, попросту игнорируют его. Эту проблему можно решить, вставив в элемент `<progress>` резервное содержимое следующим образом:

```
<progress value="0.25">25%</progress>
```

При этом следует помнить, что это резервное содержимое не будет отображаться в браузерах, которые поддерживают этот элемент.

Для индикатора выполнения есть еще одна опция в виде *неопределенного* состояния индикатора, которое указывает, что задание выполняется, но точное время его завершения неизвестно. (Индикатор в неопределенном состоянии можно рассматривать как вычурное сообщение "Задание в процессе выполнения".) Визуально индикатор в неопределенном состоянии выглядит как серое поле, вдоль которого периодически слева направо пробегает зеленая размытая полоска. Чтобы создать этот индикатор, просто не употребляйте атрибут `value`:

```
<progress>Task in progress...</progress>
```

Элемент `<meter>` имеет подобную модель, но отображает любой вид измерений. Иногда его еще называют *шкалой*. Часто значение атрибута `value` этого элемента отображает какую-то действительную величину, например денежную сумму на счету, количество дней, вес в килограммах и т. п. Отображение этой информации управляется установкой значений атрибутов `min` и `max`:

```
Your suitcase weighs: <meter min="5" max="70" value="28">
                        28 pounds</meter>
```

Как и для элемента `<progress>`, содержимое внутри элемента `<meter>` отображается только в браузерах, которые не поддерживают его. Но иногда значение элемента `<meter>` нужно вывести в числовом формате. В таком случае его нужно задать на странице обычной разметкой; при этом резервное значение внутри элемента не требуется. Этот подход демонстрируется в следующем примере. Вся информация выводится на страницу обычной разметкой, а элемент `<meter>` додается в качестве необязательного визуального ориентира:

```
<p>Our goal is to raise $50,000 for SLF (Save the
                                          Lemmings Foundation).</p>
<p>So far we've raised $14,000. <meter max="50000" value="14000"></meter>
```

Элемент `<meter>` также достаточно интеллектуальный, чтобы указывать значения выше или ниже допустимых и в то же самое время отображать их должным образом. Для этого в нем используются атрибуты `low` и `high`. Например, значение `value`, превышающее значение `high`, но меньше значения `max`, будет выше какого-то требуемого значения, но все еще в пределах допустимого максимального значения. Подобным образом значение `value` ниже значения `low`, но выше значения `min`, не удовлетворяет какому-либо требованию низкого значения, но будет все еще в пределах допустимого минимального значения.

В следующем листинге показан пример использования атрибута high:

```
Your suitcase weighs:
<meter min="5" max="100" high="70" value="79">79 pounds</meter>*
<p><small>* A surcharge applies to suitcases heavier than 70 pounds.
   </small></p>
```

Браузеры могут использовать или не использовать информацию атрибутов low и high. Например, браузер Chrome отображает слишком высокие значения (как в предыдущем примере) желтым цветом, но значения ниже low показывает как обычные. Наконец, атрибут optimum служит для указания определенного оптимального значения, но использование этого атрибута не влияет на отображение этого значения в современных браузерах.

В общем, атрибуты <progress> и <meter> предоставляют незначительные удобства, которые будут полезными, когда уровень их поддержки браузерами будет немного выше.

Элементы *<command>* и *<menu>* для создания кнопок команд и меню

Эти две возможности можно считать самыми замечательными, но пока не реализованными. Идея заключается в том, чтобы использовать один элемент для представления действия, которое пользователь может активировать (т. е. <command>), а другой элемент для группирования нескольких таких элементов (т. е. <menu>). В зависимости от конструкции и оформления элемент <menu> может принимать какие угодно формы, от панели инструментов, закрепленной на стороне окна браузера, до всплывающего меню, открывающегося при щелчке мышью где-либо на странице. Но в настоящее время ни один из браузеров не поддерживает эти элементы, поэтому нужно подождать, чтобы узнать, действительно ли они такие замечательные.

Веб-страница как HTML-редактор

Как мы узнали в *главе 1*, одним из принципов HTML5 является асфальтирование тропинок, иными словами, официальное признание частью стандарта нестандартных возможностей, которые уже широко используются веб-разработчиками. Одним из наилучших примеров следования этому принципу является включение в стандарт двух необычных атрибутов — contentEditable и designMode, которые позволяют сделать из обыкновенного браузера простой HTML-редактор.

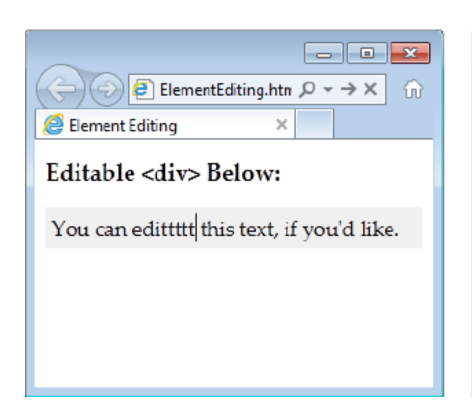
Эти два атрибута не новы. Более того, их поддержка была добавлена еще в Internet Explorer 5 на заре становления Интернета. В то время большинство разработчиков отказалось от их использования, рассматривая их всего лишь как расширения Windows для Интернета. Но с течением лет все больше браузеров начали копировать применяемый в IE практический, но причудливый подход к расширенному HTML-редактированию. В настоящее время все браузеры для настольных компьютеров поддерживают эти атрибуты, хотя они никогда и не были частью официального стандарта.

Редактирование элементов с помощью атрибута *contentEditable*

Рассмотрим первый инструмент для HTML-редактирования — атрибут `contentEditable`. Добавление этого атрибута к любому элементу позволяет редактировать содержимое этого элемента, когда оно отображается в окне браузера:

```
<div id="editableElement" contentEditable>You can edit this text,
                                          if you'd like.</div>
```

Скорее всего, вы не заметите никакой особенности в отображении этого текста в браузере. Но щелчок где-либо в отображаемом браузером тексте помещает в него курсор редактирования (рис. 4.16).



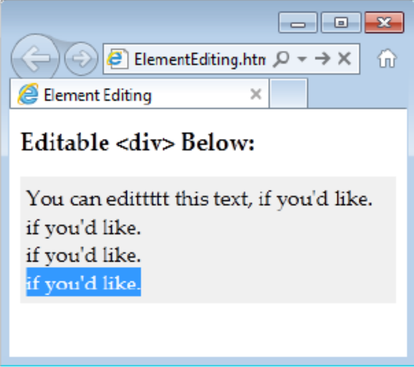


Рис. 4.16. Щелчок мышью в редактируемой области помещает в нее курсор редактирования, после чего в ней можно перемещаться с помощью клавиш со стрелками, а также добавлять и удалять текст *(слева)*. Или выделять текст, который потом можно вырезать, копировать и вставлять *(справа)*. Это немного напоминает редактирование в обычном текстовом редакторе, только курсор нельзя устанавливать за пределами области, определенной тегами `<div>`, хотя добавление текста расширяет эту область вплоть до заполнения всего окна и дальше

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Когда применять HTML-редактирование?

Прежде чем приступить к использованию расширенного HTML-редактирования, стоит узнать, а для чего оно вообще-то применяется. Несмотря на мгновенно производимое впечатление, HTML-редактирование является специализированной возможностью, которая не каждому понравится. Имеет смысл использовать его, чтобы предоставить пользователям быстрый способ изменять HTML-содержимое, например, добавлять сообщения в блоги, вносить обзоры, подавать объявления или составлять сообщения другим пользователям.

Но даже если вы решите, что вам нужны такие возможности, атрибуты `contentEditable` и `designMode` могут оказаться не лучшим выбором для их реализации. Они не предоставляют разработчику всех удобств настоящего инструмента для веб-дизайна, таких как команды для редактирования разметки, возможность просматривать и редактировать исходный HTML-код, средства проверки орфографии и т. п. Создание намного более функционального редактора с помощью расширенного HTML-редактирования *возможно*, хоть и с приложением определенных усилий. Но если вам действительно требуется функциональность расширенного редактирования, может быть, лучше вос-

пользоваться чьим-то другим уже готовым редактором, который можно вставить в свои страницы. Обзор наиболее популярных редакторов см. по адресу **http://ajaxian.com/archives/richtexteditors-compared**.

В приведенном ранее примере редактируемая область `<div>` содержала только текст. Но с такой же легкостью в нее можно вставить другие элементы. В действительности, элемент `<div>` может содержать разметку всей веб-страницы, делая ее редактируемой. Так, атрибут `contentEditable` можно применять к нескольким элементам, позволяя редактировать несколько областей окна браузера.

Совет

Некоторые браузеры поддерживают встроенные команды форматирования. Например, в Internet Explorer текст можно делать жирным, курсивом и подчеркивать с помощью комбинаций клавиш <Ctrl>+<B>, <Ctrl>+<I> и <Ctrl>+<U> соответственно. Отменить последнее редактирование в Firefox можно посредством комбинации клавиш <Ctrl>+<Z>. Все эти "горячие" клавиши можно использовать также в браузере Chrome. Дополнительную информацию об этих командах редактирования и о том, как можно создать настраиваемую панель инструментов для их инициирования, см. в статье в двух частях от разработчиков браузера Opera: **http://tinyurl.com/htmlEdit** и **http://tinyurl.com/htmlEdit2**.

Обычно атрибут `contentEditable` в разметку не включается. Вместо этого он включается с помощью JavaScript-кода и отключается по завершению редактирования. В следующем листинге приведены соответствующие функции:

```
function startEdit() {
  // Включаем редактирование элемента.
  var element = document.getElementById("editableElement");
  element.contentEditable = true;
}

function stopEdit() {
  // Отключаем редактирование элемента.
  var element = document.getElementById("editableElement");
  element.contentEditable = false;

  // Выводим редактируемый текст в окне сообщения.
  alert("Your edited content: " + element.innerHTML);
}
```

А для вызова этих функций используем соответствующие кнопки:

```
<button onclick="startEdit()">Start Editing</button>
<button onclick="stopEdit()">Stop Editing</button>
```

Только смотрите — не поместите эти кнопки в редактируемую область страницы, т. к. при редактировании страницы ее элементы прекращают генерировать события, и ваш код не сможет запуститься.

На рис. 4.17 показаны результаты применения этого кода — введена строка текста, в которой слово "YES" отформатировано жирным шрифтом посредством "горячих" клавиш <Ctrl>+<B>.

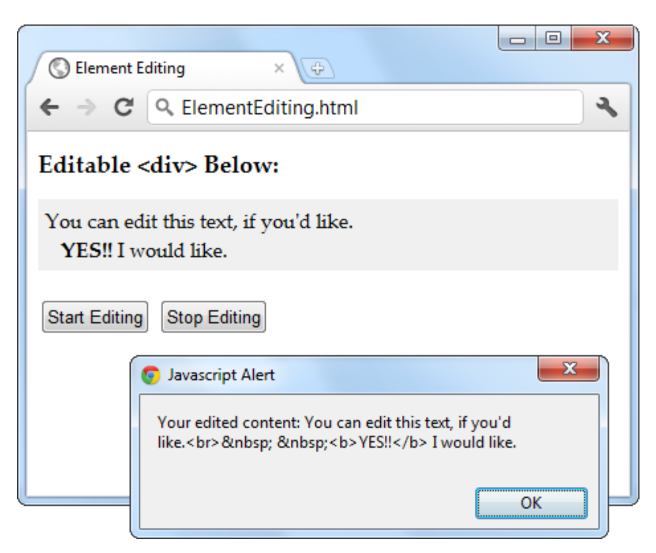


Рис. 4.17. Как видно, редактирование содержимого элемента действительно изменяет хранящееся в памяти содержимое страницы. В примере новое содержимое просто показано в окне сообщений, но в настоящей странице эти данные были бы отправлены на веб-сервер, скорее всего, с помощью объекта `XMLHttpRequest`

Примечание

HTML-редактирование работает по-разному на различных браузерах. Например, в браузере Chrome нажатие комбинации клавиш <Ctrl>+<B> добавляет элемент `<b>`, а в браузере IE — элемент `<strong>`. Подобные расхождения происходят и при нажатии клавиши <Enter> для добавления новой строки или клавиши <Backspace> для удаления тега. Поэтому логично стандартизировать возможность HTML-редактирования в HTML5, чтобы заставить разработчиков браузеров поддерживать одинаковое редактирование.

Редактирование страницы с помощью атрибута *designMode*

Атрибут `designMode` похож на атрибут `contentEditable`, с той разницей, что он позволяет редактировать всю веб-страницу. Это может показаться слегка проблематичным, ведь при редактировании страницы отключаются события элементов. Тогда как мы сможем нажимать кнопки, чтобы управлять процессом редактирования? Решение этой проблемы простое — редактируемый документ помещается внутри элемента `<iframe>`, который ведет себя, как сверхмощное окно редактирования (рис. 4.18).

Разметка этой страницы до приятного проста. В следующем листинге приведено все содержимое элемента `<body>` страницы.

```
<h1>Editable Page</h1>
<iframe id="pageEditor" src="ApocalypsePage_Revised.html"></iframe>
<div>
  <button onclick="startEdit()">Start Editing</button>
```

```
  <button onclick="stopEdit()">Stop Editing</button>
</div>

<h1>Edited HTML</h1>
<div id = "editedHTML"></div>
```

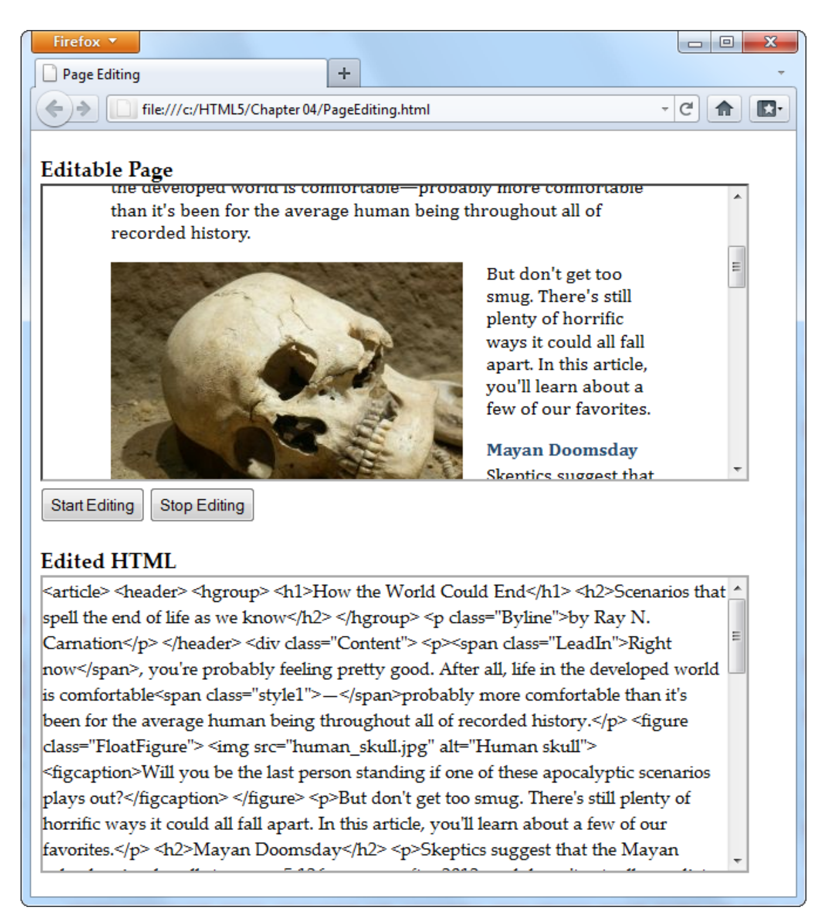


Рис. 4.18. Эта страница содержит две области. Первая область создается элементом `<iframe>` и содержит страницу примера об апокалипсисе из *главы 2*. А вторая область создана обычным элементом `<div>` и содержит HTML-разметку страницы после редактирования. Управление редактированием осуществляется посредством двух кнопок вверху страницы, с помощью которых включается и выключается атрибут `designMode` окна элемента `<iframe>`

Видно, что, как и в предыдущем примере, в этом примере используются функции `startEdit()` и `stopEdit()`. Но их код несколько изменен, чтобы управлять атрибутом `designMode`, а не `contentEditable`:

```
function startEdit() {
  // Включаем атрибут designMode элемента <iframe>.
  var editor = document.getElementById("pageEditor");
  editor.contentWindow.document.designMode = "on";
}
```

```
function stopEdit() {
  // Выключаем атрибут designMode элемента <iframe>.
  var editor = document.getElementById("pageEditor");
  editor.contentWindow.document.designMode = "off";

  // Отображаем отредактированную разметку
  // (просто чтобы убедиться, что редактирование было выполнено).
  var htmlDisplay = document.getElementById("editedHTML");
  htmlDisplay.textContent = editor.contentWindow.document.body.innerHTML;
}
```

Этот пример должен дать вам представление о возможностях HTML-редактирования. Например, щелкните на изображении, чтобы выделить его для редактирования. Теперь можно изменять размеры изображения, перетаскивать его на новое место или совсем удалить его. То же самое можно делать и с элементами управления формы, если редактируемая страница содержит их.

Конечно же, если вы хотите превратить этот пример во что-то практическое, то вам нужно будет серьезно доработать его. Прежде всего следует добавить другие элементы управления редактированием. Если вы хотите получше разобраться в модели команд, в этом вам опять помогут разработчики браузера Opera (см. **http://tinyurl.com/htmlEditl** и **http://tinyurl.com/htmlEdit2**). Вторым делом нужно будет делать что-то полезное с отредактированной разметкой, например, отправить ее на сервер, используя объект `XMLHttpRequest` *(см. разд. "Объект XMLHttpRequest" главы 11)*.

Еще одно предостережение. Этот пример не будет запускаться с локального жесткого диска на всех браузерах. Internet Explorer и Chrome заблокируют его по причинам безопасности, но на Firefox не будет никаких проблем. Чтобы избежать возможных проблем, его можно запустить с сайта книги: **www.prosetech.com/html5**.

5

ГЛАВА 5

Аудио и видео

Было время, когда Интернет использовался в основном для обмена данными научных исследований. Потом было время перемен, и в мгновение ока Интернет стал одним из двигателей новостной индустрии и торговли. Еще несколько мгновений, и вот мы уже здесь, с Интернетом, использующим новейшие сетевые технологии для доставки на дом по всему миру невероятнейших объемов видеоинформации самого разного содержимого от трансляций с марсохода в режиме реального времени до дешевой комедии.

Важность этого сдвига трудно переоценить. Проект, который в начале 2005 г. был на стадии "разве не было бы это замечательно", теперь реализовался в виде YouTube. Видеосюжеты продолжительностью в 3—4 минуты наводнили Интернет. И, как докладывает гигант сетевого оборудования и услуг Cisco, эта тенденция не замедляется, и, согласно оценкам этой компании, к 2013 г. ошеломляющие 90% всего интернет-трафика будет составлять видео.

Поразительно, это грандиозное изменение произошло несмотря на то, что HTML и браузеры не обладают встроенной поддержкой видео и даже аудио. Вместо этого они полагаются на подключаемые модули, такие как Flash, которые удовлетворяют надобности большинства пользователей в большинстве случаев. Но в этом HTML-видеопокрытии есть очевидные мертвые зоны, наподобие той, которую создал iPad компании Apple, не поддерживающий Flash.

Стандарт HTML5 пытается решить эти пробелы введением элементов `<audio>` и `<video>`, которых так не хватало в HTML все эти годы. Наконец, содержимое мультимедиа получило единообразную, стандартную поддержку, не требующую подключаемых модулей. Но не все в этой истории так гладко. Основные разработчики браузеров сцепились в битве аудио- и видеоформатов, которая намного грязнее, чем война форматов Blu-Ray и HD-DVD. Печальным последствием этой битвы является отсутствие единого аудио- и видеоформата, который бы работал на всех браузерах, и чтобы файлы мультимедиа можно было бы воспроизводить в HTML, их нужно кодировать по-разному для разных браузеров. В этой главе мы узнаем, как это делается.

Но сначала, давайте сделаем шаг назад и рассмотрим общую картину, чтобы увидеть, в каком состоянии находилось воспроизведение видео перед тем, как за это взялся HTML5.

Основные сведения о воспроизведении видео в современных программах

Не прибегая к HTML5, видео в веб-страницу можно добавить двумя способами. Самый простой состоит в использовании элемента `<embed>`. Потом браузер создаст видеоокно под проигрыватель Windows Media Player, Apple QuickTime или какой-либо другой проигрыватель и разместит его на странице.

Основная проблема с этим подходом заключается в том, что он полностью отдает разработчика на милость поддержки браузера. Разработчик не властен управлять воспроизведением, может не иметь возможности буферизировать видео, чтобы предотвратить задержки с воспроизведением, а также не знает, сможет ли данный видеофайл воспроизводиться в разных браузерах или операционных системах.

Второй подход заключается в использовании подключаемого модуля браузера, наподобие сравнительно недавнего новичка в этой области Silverlight корпорации Microsoft или старожила Flash от компании Adobe. До недавнего времени использование модуля Flash полностью решало задачу поддержки видеосодержимого браузерами. Ведь видеоформат Flash работает всюду, где установлен модуль Flash, что в настоящее время составляет свыше 90% подключенных к Интернету компьютеров. Технология Flash также предоставляет почти неограниченный контроль над воспроизведением видео. Разработчик может, например, использовать готовый видеоплеер Flash сторонней фирмы или создать собственный и индивидуально оформить каждую кнопку управления.

Но подход с использованием Flash также не идеальный. Чтобы вставить видео Flash в веб-страницу, в нее нужно вставить определенный объем безобразной разметки, содержащей элементы `<object>` и `<embed>`. Кроме этого, видеофайлы для показа нужно закодировать в требуемый формат, а также может потребоваться приобрести дорогостоящий инструментарий Flash-разработки и научиться пользоваться им, что может потребовать серьезных усилий. Но хуже всего это новая волна мобильных устройств компании Apple — iPhone и iPad. Эти устройства органически не воспринимают Flash и выводят пустую рамку в том месте страницы, где вставлено это видео.

Примечание

Модули расширения также имеют свойство временами сбоить. Причина кроется в принципе их работы. Например, при посещении страницы, на которой используется видео Flash, браузер позволяет модулю Flash взять под контроль прямоугольную область где-нибудь на странице. В большинстве случаев этот подход работает нормально, но незначительные ошибки или нестандартные системные настройки могут вызвать неожиданные помехи и сбои, такие как, например, искаженное видео или потребление веб-страницей огромных объемов памяти, в результате чего картинка начинает ползти, как улитка.

Тем не менее, если вы сегодня просматриваете видео в Интернете (но не с помощью iPhone или iPad), то, скорее всего, это видео обернуто в мини-приложение Flash. Если вы не уверены, щелкните на видеоплеере правой кнопкой мыши. Если откроется меню с командой наподобие **О программе Adobe Flash Player 10**, тогда вы наверняка имеете дело с вездесущим модулем Flash. И даже при переходе на HTML5 разработчикам, скорее всего, понадобится резервное решение с использованием Flash для браузеров-старожилов, которые остались в прошлом, таких как, например, Internet Explorer 8.

Примечание

В 2010 г. сервис YouTube ввел в действие HTML-видеопроигрыватель на испытательной основе. Попробовать этот проигрыватель можно, посетив страницу **www.youtube.com/html5**. Но во всех других случаях YouTube полагается исключительно на Flash.

Представляем видео и аудио HTML5

Поддержка видео и аудио HTML5 основана на простой идее. Точно так же, как с помощью элемента `<img>` в веб-страницу можно вставлять изображения, в нее должно быть возможным вставить звук посредством элемента `<audio>` и видео с помощью элемента `<video>`. Вполне логично, HTML5 позволяет вставлять оба эти типа мультимедиа.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Поворачивайте назад сейчас, если...

К сожалению, некоторые аспекты воспроизведения мультимедиа находятся вне досягаемости новых аудио- и видеовозможностей HTML5. Для следующего содержимого и/или способов его воспроизведения нужно опять обращаться к Flash (по крайней мере, на данный момент).

- **Лицензированное содержимое.** Видеофайлы HTML5 не используют никакой системы для защиты от копирования. По сути, народ может загружать HTML5-видео с такой же легкостью, как и изображения — просто щелкнув правой кнопкой мыши по видео и выбрав опцию **Сохранить**.
- **Потоковое аудио или видео.** В HTML5 нет способа для передачи аудио или видео от одного компьютера к другому в потоковом режиме. Поэтому, разработчикам чат-программ, в которых посетители веб-страницы используют микрофон и/или веб-камеру, придется продолжать работать с Flash. Разработчики HTML5 экспериментируют с элементом `<device>`, который может предоставить эту возможность, но в настоящее время решения с использованием только HTML5 нет ни для какого браузера.
- **Адаптивное потоковое видео.** Для продвинутых веб-сайтов с большими объемами видеосодержимого, наподобие YouTube, требуется многоуровневое управление буферизацией и пересылкой видеопотока. Им нужно предоставлять видео в разных разрешениях, видеособытий реального времени, а также настраивать качество видео под пропускную возможность интернет-подключения пользователя. До тех пор пока HTML5 не сможет предоставлять эти возможности, видеообменные сайты могут добавить поддержку HTML5 только как опцию, но полностью переходить на него с Flash не будут.

- **Высококачественное аудио с низкой задержкой.** Некоторые приложения требуют начинать воспроизведение аудио без задержки или проигрывать несколько аудиоклипов с идеальной синхронизацией. В качестве примеров таких приложений можно назвать виртуальный синтезатор, музыкальный визуализатор или игру реального времени с множеством накладывающихся звуковых эффектов. И в то время как разработчики браузеров усиленно работают над улучшением HTML5-аудиопроизводительности, оно еще не отвечает этим требованиям.
- **Динамическое создание или редактирование аудио.** А если вам нужно не только воспроизводить записанное аудио, но также анализировать аудиоинформацию, модифицировать или создавать аудио — и все это в режиме реального времени? Новые стандарты, такие как Audio Data API, разрабатываемые под спонсорством Firefox, состязаются в добавлении возможностей этого типа к HTML5-аудио, но в настоящее время они еще не доступны.

Воспроизведение аудио с помощью элемента *<audio>*

В следующем коде приведен простейший пример использования элемента `<audio>`:

```
<p>Hear us rock out with our new song,
<cite>Death to Rubber Duckies</cite>:</p>
<audio src="rubberduckies.mp3" controls></audio>
```

Атрибут `src` содержит имя аудиофайла для воспроизведения, а атрибут `controls` указывает браузеру, что нужно отобразить базовые элементы управления воспроизведением. Своим внешним видом эти элементы управления слегка отличаются от браузера к браузеру, но все они имеют одинаковое назначение: разрешают пользователю начинать и останавливать воспроизведение, переходить в другое место записи и регулировать громкость (рис. 5.1).

Примечание

Элементы `<audio>` и `<video>` должны иметь как открывающий, так и закрывающий тег. Использование синтаксиса пустых элементов (например, `<audio/>`) не допускается.

Кроме атрибута `controls` элемент `<audio>` поддерживает еще три атрибута: `preload`, `auotoplay` и `loop`. Атрибут `preload` указывает браузеру способ загрузки аудиофайла. Значение `auto` этого атрибута указывает браузеру загружать аудиофайл полностью, чтобы он был доступен, когда пользователь нажмет кнопку воспроизведения. Конечно же, загрузка осуществляется в фоновом режиме, чтобы посетитель веб-страницы мог перемещаться по странице и просматривать ее, не дожидаясь завершения загрузки аудиофайла.

Атрибут `preload` может принимать два значения. Значение `metadata` указывает браузеру загрузить первую небольшую часть файла, достаточную, чтобы определить некоторые его основные характеристики (например, общий размер файла). Атрибут `none` указывает браузеру не загружать аудиофайл автоматически. Эти опции можно использовать для того, чтобы сэкономить пропускную способность подключения, например, если страница содержит большое число элементов

<audio>, но ожидается, что пользователь будет проигрывать лишь некоторые из них:

```
<audio src="rubberduckies.mp3" controls preload="metadata"></audio>
```

Когда атрибуту `preload` задано значение `none` или `metadata`, загрузка аудиофайла начинается после того, как пользователь нажмет кнопку воспроизведения. К счастью, браузеры могут без проблем проигрывать одну часть аудиофайла, в то время как загружать другую, если только интернет-подключение не слишком медленное.

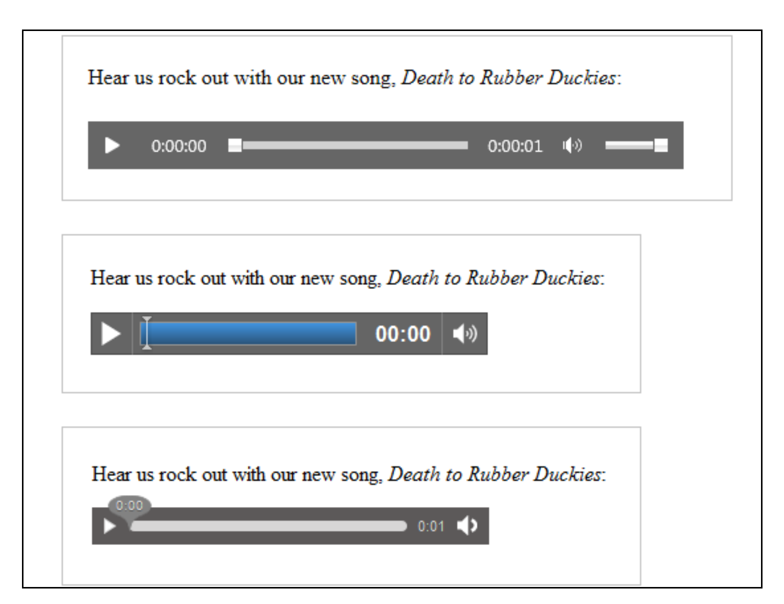


Рис. 5.1. Так выглядят элементы управления воспроизведением в трех разных браузерах: Internet Explorer *(вверху)*, Google Chrome *(в центре)* и Firefox *(внизу)*. Но чтобы создать страницу с аудио, которое может проигрываться на всех трех браузерах, потребуется немного поколдовать над аудиоформатами

Если значение атрибута `preload` не установлено, то браузеры действуют по своему индивидуальному усмотрению. Большинство браузеров предполагает `auto` в качестве значения по умолчанию, но в Firefox это `metadata`. Кроме этого, важно отметить, что атрибут `preload` не является обязательным для выполнения правилом, а рекомендацией браузеру желаемого действия, которую он может игнорировать в зависимости от других обстоятельств. А некоторые устаревшие браузеры вообще не обращают внимания на атрибут `preload`.

Примечание

Если вставить в разметку несколько элементов <audio>, то браузер создаст отдельную полосу элементов управления воспроизведением для каждого из них. Посетитель веб-страницы может прослушивать аудио на одном из них или на всех сразу.

Атрибут `autoplay` указывает браузеру начать воспроизведение сразу же после завершения загрузки страницы:

```
<audio src="rubberduckies.mp3" controls autoplay></audio>
```

Если этот атрибут не используется, пользователь должен нажать кнопку запуска, чтобы начать воспроизведение.

Элемент `<audio>` можно использовать для того, чтобы проигрывать фоновую музыку или обеспечить звуковые эффекты в игре с браузерным интерфейсом. Для этого из него нужно убрать атрибут `controls` и вставить атрибут `autoplay` (или же осуществлять воспроизведения посредством кода JavaScript, как рассматривается в *разд. "Управление плеером посредством JavaScript" далее в этой главе*). Но будьте осторожны в применении этого подхода и не забывайте, что для такой страницы все равно требуется какой-либо способ для прекращения воспроизведения.

> ***Внимание!***
>
> Вряд ли кому понравится страница, с которой несется оглушительная музыка или звуковые эффекты, и нет возможности отключить звук. Если вы решите использовать элемент `<audio>` без атрибута `controls`, то должны непременно добавить кнопку, которая посредством JavaScript отключает звук.

Наконец, атрибут `loop` указывает браузеру повторять воспроизведение:

```
<audio src="rubberduckies.mp3" controls loop></audio>
```

Воспроизведение в большинстве браузеров достаточно плавное, что позволяет создать с помощью этого метода непрерывную повторяющуюся звуковую дорожку. Секрет состоит в том, чтобы использовать повторяемое аудио, с одинаковым началом и окончанием. На сайте **www.flashkit.com/loops** можно найти сотни бесплатных образцов таких аудиофайлов. (Эти повторяющиеся аудиофайлы были изначально предназначены для использования с Flash-плеерами, но также имеются в форматах MP3 и WAV.)

Если элемент `<audio>` выглядит так хорошо, что даже не верится, к сожалению, так оно и есть. В *разд. "Войны форматов и резервные решения" далее в этой главе* мы рассмотрим проблемы с аудиоформатами, которые ввергают HTML5-разработчиков в депрессию. Но сначала стоит познакомиться с близким родственником элемента `<audio>` — элементом `<video>`.

Воспроизведения видео с помощью элемента *<video>*

С элементом `<audio>` хорошо идет в паре элемент `<video>`. Он применяет такие же атрибуты `src`, `controls`, `autoplay` и `loop`. Пример использования этого элемента показан в следующем коде:

```
<p>A butterfly from my vacation in Switzerland!</p>
<video src="butterfly.mp4" controls></video>
```

Как и в случае с элементом `<audio>`, атрибут `controls` указывает браузеру создать набор элементов управления воспроизведением (рис. 5.2). В большинстве браузеров эти элементы скрываются при щелчке где-нибудь в области страницы и отображаются опять при наведении курсора мыши на область видеоплеера.

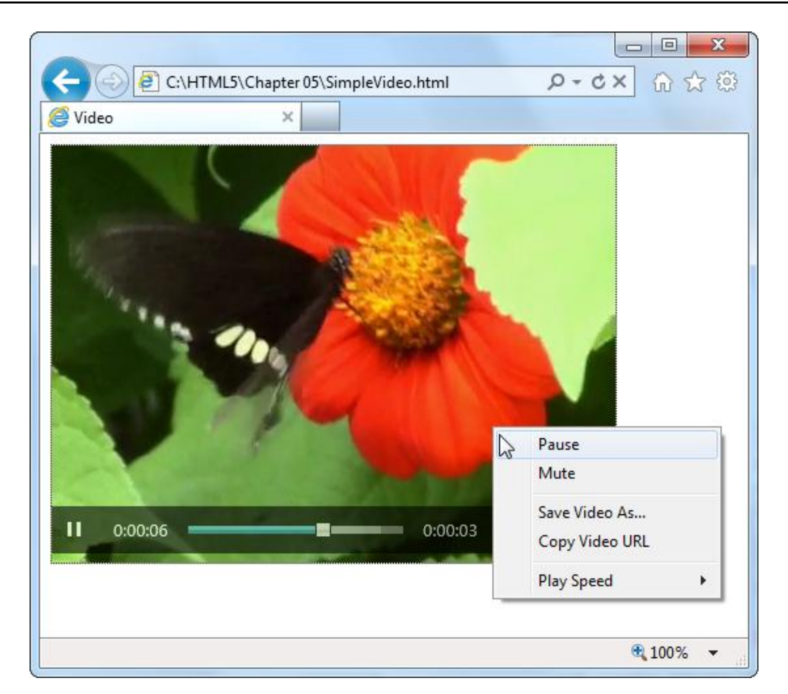


Рис. 5.2. Проигрыватель элемента `<video>` можно легко принять за проигрыватель Flash. Но если щелкнуть правой кнопкой мыши по проигрывателю, то откроется более простое контекстное меню, содержащее опцию сохранения видеофайла на жесткий диск локального компьютера. В зависимости от браузера, это контекстное меню также может содержать команды для изменения скорости воспроизведения, повтора, полноэкранного режима и выключения звука

Кроме общих с элементом `<audio>` атрибутов, элемент `<video>` имеет три своих собственных атрибута: `height`, `width` и `poster`.

Атрибуты `height` и `width` устанавливают высоту и ширину окна воспроизведения в пикселах, соответственно. Следующий код показывает пример создания области воспроизведения размером 400×300 пикселов:

```
<video src="butterfly.mp4" controls width= "400" height="300"></video>
```

Размеры окна воспроизведения должны совпадать с размером видео, но лучше явно указать их, чтобы оформление страницы не искажалось до того, как видеофайл загрузится (или если видеофайл вовсе не загружается).

Наконец, атрибут `poster` позволяет указать изображение, которое можно использовать вместо видео:

```
<video src="butterfly.mp4" controls poster="swiss_alps.jpg"></video>
```

Браузеры показывают это изображение в трех ситуациях: когда первый кадр видео еще не загрузился, атрибуту `preload` присвоено значение `none` или указанный видеофайл отсутствует.

Хотя на данном этапе мы рассмотрели все, относящееся к аудио- и видеоразметке, эти возможности можно значительно расширить с помощью стратегически размещенного кода JavaScript. Но прежде чем мы углубимся в изучение элементов

`<audio>` и `<video>`, нам нужно разобраться с проблемами поддержки аудио- и видеокодеков.

> **ПРИМЕЧАНИЕ**
>
> Элементы `<audio>` и `<video>` имеют еще два атрибута. Атрибут `muted` выключает звук сразу же при загрузке страницы. Пользователь может включить звук обратно с помощью элементов управления воспроизведением. А атрибут `mediagroup` используется для связки нескольких мультимедийных файлов, чтобы синхронизировать их воспроизведение. Эта возможность особенно полезна, когда звуковая дорожка видео находится в отдельном аудиофайле. Но в настоящее время браузеры не поддерживают ни один из этих атрибутов.

Войны форматов и резервные решения

В рассмотренных ранее примерах использовались два популярных стандарта: MP3 для аудио и H.264 для видео. Этого достаточно для Internet Explorer и Safari, но не для других браузеров (рис. 5.3).

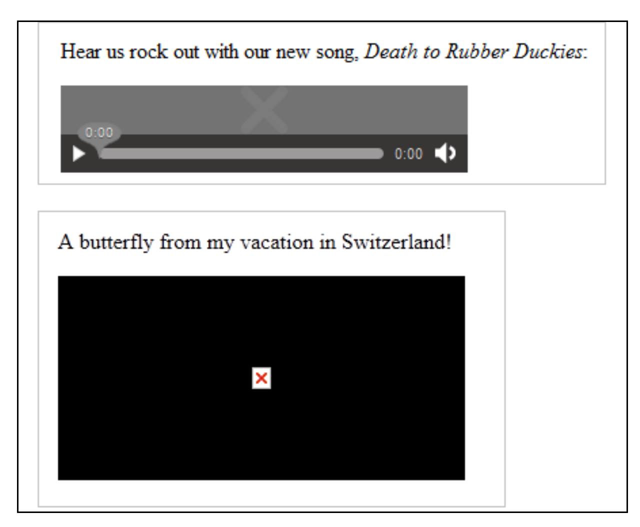


Рис. 5.3. Результаты попытки воспроизвести MP3-файл в Firefox *(вверху)* и видеофайл в Internet Explorer *(внизу)*

По поводу войны мультимедийных форматов для HTML5 у веб-разработчиков имеется несколько сердитых вопросов. Таких как, действительно ли аудио и видео HTML5 находятся в состоянии безнадежной конфронтации и на ком лежит главная вина за это? Но на эти вопросы нет ясных и однозначных ответов. У каждого разработчика браузеров есть свои оправдания и объяснения, каким стандартам мультимедиа отдать предпочтение. Небольшие разработчики (такие как Mozilla, создатели браузера Firefox) не желают платить непомерно высокую для них цену за лицензию на использование таких популярных стандартов, как MP3 для аудио или H.264 для видео. И их трудно винить за это, ведь они предоставляют свои продукты бесплатно.

У компаний покрупнее (таких как Microsoft или Apple) имеются свои оправдания, почему надо избегать нелицензированных стандартов. Они жалуются, что качество работы этих стандартов будет ниже (в настоящее время они не поддерживают аппаратное ускорение) и что они не так широко используются, как запатентованные стандарты, такие как, например, H.264, который используется в камкордерах, проигрывателях Blu-Ray и во многих других разных устройствах. Но самая большая проблема может состоять в том, что никто по-настоящему не уверен, что эти нелицензированные стандарты не связаны с чьей-либо интеллектуальной собственностью. Если такие связи имеются, используя эти стандарты, крупные компании, наподобие Microsoft или Apple, делают себя уязвимыми к дорогостоящим искам за нарушение патентных прав, которые могут тянуться годами.

ЧАСТО ЗАДАВАЕМЫЙ ВОПРОС
Лицензирование стандарта H.264

Я использую стандарт H.264 для своего видео.
Должен ли я приобрести лицензию для этого?

Если вы используете в своем продукте декодер H.264 (например, вы разрабатываете браузер, который может воспроизводить видео в стандарте H.264), то определенно должны приобрести лицензию. Но если вы предоставляете видео для воспроизведения, то здесь не все так ясно и однозначно.

Сначала хорошие новости. Если вы используете стандарт H.264 для создания бесплатных видео, то вам никогда ничего ни за какую лицензию платить будет не нужно. Если вы создаете видео коммерческого направления, но которые в действительности не продаются (например, рекламный ролик или продвигаете себя в интервью), то здесь тоже платить ничего не нужно.

Но если вы продаете на своем веб-сайте видеосодержимое в формате H.264, вам, может быть, придется уплатить лицензионный сбор фирме MPEG-LA либо сейчас, либо в будущем. В настоящее время все зависит от количества подписчиков. Если их у вас меньше чем 100 тысяч, платить не нужно, но если от 100 до 250 тысяч, то вас попросят раскошелиться на $25 000 в год. Эта сумма может показаться не такой и значительной для компании с таким количеством подписчиков, и она может быть ничтожной по сравнению с другими расходами, такими как, например, стоимость профессиональных инструментов для кодирования. Но эти числа могут измениться при пересмотре условий лицензирования в 2016 г. Крупным компаниям, которые хотят получить хорошую прибыль в области интернет-видео, может быть, предпочтительнее использовать какой-либо открытый, нелицензионный стандарт наподобие Theora или WebM.

Все юридические подробности по лицензированию стандарта H.264 можно узнать на сайте **www.mpegla.com/main/programs/AVC/Pages/Intro.aspx**.

Знакомимся с форматами

Официальный стандарт HTML5 не требует, чтобы браузеры поддерживали какой-либо конкретный аудио- или видеоформат. (Ранние версии стандарта содержали такую рекомендацию, но в результате интенсивного лоббирования она была удалена.) Вследствие этого разработчики браузеров могут выбирать форматы, которые они хотят поддерживать, несмотря на то обстоятельство, что разные форматы органически несовместимы друг с другом. Список и краткое описание основных форматов, используемых в настоящее время, приведен в табл. 5.1.

***Таблица 5.1.** Некоторые аудио- и видеостандарты, поддерживаемые HTML5-браузерами*

Формат	**Описание**	**Расширение файла**	**Тип MIME**
MP3	Самый популярный аудиоформат в мире. Но стоимость лицензии делает его непрактичным для небольших разработчиков, наподобие Firefox и Opera	mp3	`audio/mp3`
Ogg Vorbis	Открытый, бесплатный стандарт, предоставляющий высококачественное сжатое аудио, сравнимое с MP3	ogg	`audio/ogg`
WAV	Первоначальный формат для сырого цифрового аудио. Не использует сжатие, поэтому файлы невероятно большого объема и непригодны для большинства интернет-приложений	wav	`audio/wav`
H.264	Промышленный стандарт для кодирования видео, особенно при работе с видео высокой четкости. Применяется в бытовых устройствах (таких как проигрыватели и камкордеры Blu-Ray), на видеообменных сайтах (таких как YouTube и Vimeo) и в браузерных модулях расширения (таких как Flash и Silverlight)	mp4	`video/mp4`
Ogg Theora	Открытый, бесплатный стандарт для видео, созданный разработчиками аудиостандарта Vorbis. Побайтно, качество и производительность ниже стандарта H.264, но достаточно высокие, чтобы удовлетворить потребности большинства пользователей	ogv	`video/ogg`
WebM	Новейший бесплатный видеоформат, созданный Google на основе приобретенного ими VP8. Критики доказывают, что его качество еще не на уровне видео H.264 и он может содержать скрытые связи с другими патентами, что может вызвать лавину судебных исков в будущем. Тем не менее, WemM является наилучшим выбором для будущего открытого видео	webm	`video/webm`

В табл. 5.1 также указаны рекомендуемые расширения файлов для мультимедиа. Чтобы осознать, почему это важно, нужно понимать, что для создания видеофайла в действительности применяются три разных стандарта. Первым, наиболее очевидным, стандартом является *видеокодек*, применяемый для сжатия видео в поток данных. В качестве примера можно назвать такие кодеки, как H.264, Theora и WebM. Вторым является *аудиокодек*, который применяется для сжатия одной или нескольких аудиодорожек. Например, для видео в формате H.264 обычно используется звук в формате MP3, а для видео Theora — звук Vorbis. Наконец, *формат контейнера* применяется для упаковки видео и аудио вместе с описательной информацией и, необязательно, другими безделушками типа изображений и субтитров. Часто расширение файла обозначает формат контейнера, т. е. расширение mp4 означает контейнер типа MPEG-4, расширение ogv — контейнер Ogg и т. п.

Но не все так просто, т. к. формат контейнера поддерживает несколько разных аудио- и видеостандартов. Например, популярный контейнер Matroska (mkv) может содержать видео в формате H.264 или Theora. Чтобы не усложнять этот вопрос излишними подробностями, в табл. 5.1 каждый видеоформат соотнесен с наиболее употребляемым для его упаковки контейнером, для которого также обеспечивается наиболее высокий уровень поддержки для Интернета.

В табл. 5.1 также указан требуемый тип MIME, который нужно установить в настройках вашего веб-сервера. Если не указать правильный тип MIME, браузеры могут заупрямиться с воспроизведением вполне качественного мультимедийного файла. (Если вы не знаете, что такое типы MIME и как их настраивать, *см. врезку "На профессиональном уровне. Основные сведения о MIME-типах" далее в этой главе*.)

Поддержка браузерами форматов мультимедиа

Все аудио- и видеоформаты в мире будут вам бесполезны, если вы не знаете, как они поддерживаются разными браузерами. Разобраться в этом вопросе вам помогут табл. 5.2, в которой показана поддержка основными браузерами аудиоформатов, и табл. 5.3 — видеоформатов.

Таблица 5.2. *Поддержка браузерами аудиоформатов HTML5*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
MP3	9	—	5	3.1	—	3	—
Ogg Vorbis	—	3.6	5	—	10.5	—	—
WAV	—	3.6	8	3.1	10.5	—	—

Таблица 5.3. *Поддержка браузерами видеоформатов HTML5*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
H.264	9	—	*	3.1	—	4**	2.3
Ogg Theora	—	3.5	5	—	10.5	—	—
WebM	***	4	6	—	10.6	—	2.3

* В настоящее время браузер Chrome поддерживает этот стандарт, но разработчики обязались удалить эту поддержку в будущих версиях с целью лучшего продвижения стандарта WebM.

** Операционная система iOS 3.x поддерживает видео, но браузер Safari содержит несколько неочевидных ошибок обработки видео. Например, он отказывается проигрывать видео с установленным атрибутом `poster` *(см. разд. "Воспроизведение видео с помощью элемента <video>" ранее в этой главе)*.

*** Internet Explorer поддерживает формат WebM, но при условии, что пользователь установит требуемый кодек.

Поддержка этих форматов мобильными браузерами представляет особый вид проблем. Прежде всего, это нерегулярность работы. Некоторые функции, такие как автоматическое воспроизведение и повтор, могут не поддерживаться, а некоторые устройства могут воспроизводить видео только в специализированном проигрывателе, а не прямо в окне на веб-странице. А еще видео для мобильных устройств обычно нужно кодировать с кадром меньшего размера и худшего качества.

Совет

Если вы хотите, чтобы видео проигрывалось на мобильных устройствах, примите за правило кодировать его в формате H.264 Baseline Profile (а не в формате High Profile). Для телефонов iPhone и под управлением операционной системы Android следует использовать размер 640×480, а для BlackBerry — 480×360. Многие программы кодирования *(см. врезку "На профессиональном уровне. Кодирование файлов мультимедиа" далее в этой главе)* имеют предварительные настройки, с помощью которых можно создать видео, оптимизированное для мобильных устройств.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Основные сведения о MIME-типах

MIME-тип (также иногда называется *типом содержимого*) — это единица информации, которая идентифицирует тип содержимого веб-ресурса. Например, MIME-тип веб-страницы — `text/html`.

Перед тем как отправлять ресурс браузеру, веб-сервер отправляет ему MIME-тип этого ресурса. Например, если браузер запросил страницу SuperVideoPlayerPage.html, веб-сервер сначала отправляет ему MIME-тип `text/html`, несколько других блоков информации, а потом собственно содержимое файла. MIME-тип указывает браузеру, как ему обрабатывать следующее за ним содержимое. Таким образом, браузеру не нужно определять тип содержимого по расширению файла или каким-либо другим характеристикам.

Заботиться о MIME-типе распространенных типов файлов, например HTML-страниц и изображений, нет необходимости, т. к. все веб-серверы предоставляют такие файлы должным образом. Но в настройках некоторых веб-серверов MIME-типы для аудио и видео могут быть не указаны. Это будет проблемой, т. к. получив от сервера мультимедийный файл с неправильным MIME-типом, браузер будет сбит с толку и, скорее всего, откажется воспроизводить файл вообще.

Во избежание этой проблемы необходимо установить в настройках веб-сервера MIME-типы, перечисленные в табл. 5.1, и использовать соответствующие расширения для предоставляемых аудио- и видеофайлов. (Бессмысленно устанавливать требуемые MIME-типы, а потом использовать неправильные расширения файлов, т. к. веб-сервер должен сопоставить их вместе. Например, если настроить веб-сервер на использование MIME-типа `video/mp4` с mp4-файлами, а потом присвоить видеофайлам расширение mpFour, веб-сервер не будет иметь ни малейшего понятия, что вы от него хотите.)

Настройка MIME-типов не представляет никакой сложности, но точные действия зависят от конкретного веб-хостинга (или программного обеспечения для веб-сервера, если у вас собственный хостинг). Многие веб-хостинги используют популярную панель интерфейса cPanel. Если ваша хостинговая компания тоже использует эту панель, найдите и щелкните по значку **MIME Types**. Откроется страница, наподобие показанной на рис. 5.4, на которой можно установить требуемые параметры. Если вы сомневаетесь в своих способностях выполнить эту задачу, обратитесь за помощью к персоналу своей хостинговой компании.

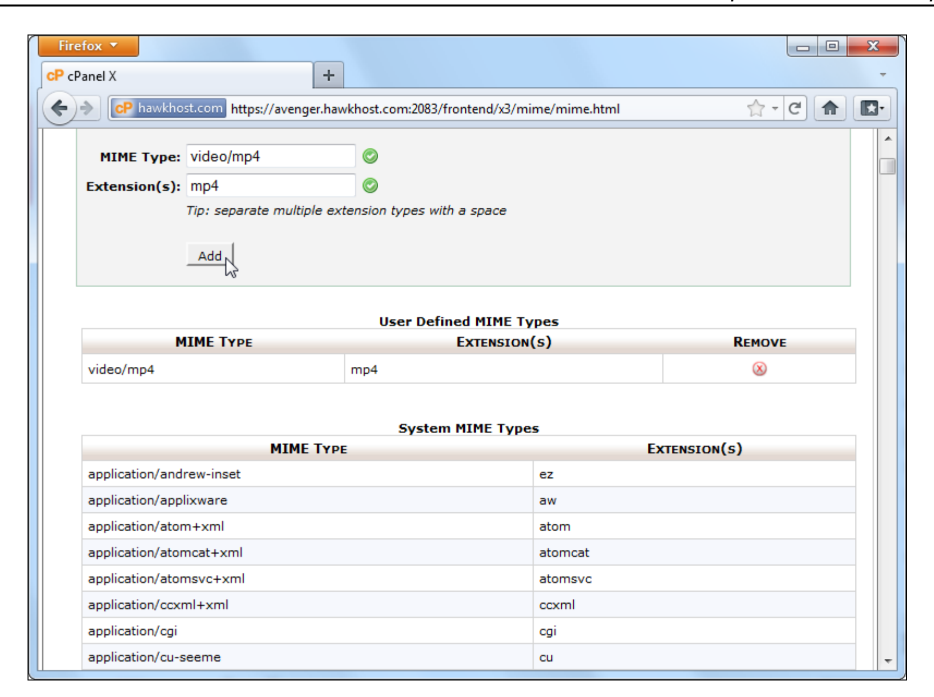


Рис. 5.4. Добавление MIME-типа для поддержки видеофайлов H.264. Во многих случаях веб-сайт будет уже настроен должным образом, и вам не нужно будет самому выполнять эти настройки

Множество форматов: как понравиться всем браузерам

Что делать бедному веб-разработчику со всеми этими форматами? Горькая правда состоит в том, что ни один аудио- или видеоформат не будет работать на всех браузерах. Если вы хотите поддерживать все браузеры, а поддерживать их все вы должны, вам нужно запастись мультимедийными файлами в нескольких форматах. Кроме этого, вам, скорее всего, нужно будет организовать резервное решение Flash для посетителей, которые пользуются браузерами, не признающими HTML5, такими как, например, IE 8.

К счастью, элементы `<audio>` и `<video>` поддерживают достаточно хорошую систему предоставления резервных решений, которая была хорошо отлажена новаторами веб-технологий. Но, к сожалению, война форматов означает, что содержимое нужно будет кодировать, по крайней мере, дважды, что является затратным процессом по времени, процессорным ресурсам и дисковому пространству.

Но прежде чем приступать к работе, нужно определиться со стратегией поддержки браузеров, которые не признают HTML5. По большому счету, веб-разработчики имеют на выбор два хороших пути.

- **Использовать Flash в качестве основного решения, а HTML5-решение — в качестве резервного.** Таким образом, все посетители вашего сайта смогут использовать Flash, за исключением тех, на чьих браузерах этот модуль не уста-

новлен. Эта стратегия имеет смысл, если вы уже предоставляете на своем сайте видеосодержимое посредством Flash, но хотите еще привлечь пользователей iPad и iPhone.

- **Использовать HTML5 в качестве основного решения, а Flash-решение — в качестве резервного.** Таким образом, все посетители получают HTML5-видео и/или аудио, за исключением тех, кто использует старые браузеры, которые получают Flash-содержимое. Если вы пойдете этим путем, можно также поддерживать меньшее число форматов HTML5. В таком случае посетители, чьи браузеры хотя и поддерживают HTML5-мультимедиа, но не поддерживают предоставляемые вами форматы, также получат Flash-содержимое. Так как будущее за этим подходом, то он является предпочтительным при условии, что текущие ограничения HTML5 видео и аудио *(см. врезку "На профессиональном уровне. Поворачивайте назад сейчас, если..." ранее в этой главе)* — вам не помеха.

В следующих разделах мы будем воплощать второй подход в жизнь. Таким образом, мы обеспечим для браузеров чисто HTML5-решение во всех случаях, когда это возможно.

Элемент <*source*>

Первым шагом в обеспечении поддержки нескольких форматов будет удаление атрибута `src` из элемента `<video>` или `<audio>` и замена его вложенным списком элементов `<source>`. Например:

```
<audio controls>
  <source src="rubberduckies.mp3" type="audio/mp3">
  <source src="rubberduckies.ogg" type="audio/ogg">
</audio>
```

В данном случае элемент `<audio>` содержит два элемента `<source>`, каждый из которых указывает на отдельный аудиофайл. Из указанных файлов браузер выбирает первый, формат которого он поддерживает. В частности, Firefox и Opera возьмут файл rubberduckies.ogg, а Internet Explorer, Safari и Chrome — файл rubberduckies.mp3.

Теоретически, браузер может определить, поддерживает он или нет конкретный файл, загрузив и исследовав небольшую его часть. Но лучшим подходом будет использовать атрибут `type`, чтобы предоставить правильный MIME-тип *(см. врезку "На профессиональном уровне. Основные сведения о MIME-типах" ранее в этой главе)*. Таким образом, браузер попытается загрузить только тот файл, который он, как считает, может воспроизвести. (Чтобы определить правильный MIME-тип, см. информацию в табл. 5.1.)

Этот же метод применяется и для элемента `<video>`. В следующем листинге показан пример указания видеосодержимого в двух разных форматах — H.264 и Theora:

```
<video controls width="700" height="400">
  <source src="beach.mp4" type="video/mp4">
  <source src="beach.ogv" type="video/ogg">
</video>
```

В этом примере следует отметить одну новую особенность. При использовании разных видеоформатов файл в формате H.264 всегда должен быть в списке первым. В противном случае он не будет проигрываться на старых устройствах iPad под управлением iOS 3.*x*. (Эта проблема была решена в операционной системе iOS 4, но размещение файла H.264 вверху списка ничем ничему не вредит.)

Примечание

Само обстоятельство, что браузер считает, будто он поддерживает определенный тип аудио или видео, не обязательно означает, что он может воспроизводить его. Например, файл может быть закодирован с сумасшедшей частотой дискретизации или с применением неизвестного кодека, но упакован в контейнер знакомого браузеру формата. Подобные проблемы можно решить, предоставив информацию о типе содержимого и кодеке в атрибуте `type`, но это внесет беспорядок в разметку.

Так сколько же видеоформатов следует использовать? Чтобы прикрыть все тылы, необходимо использовать форматы H.264 и Theora для основного решения HTML5 и Flash для резервного (которое рассматривается в следующем разделе). Для лучшего качества видео формат Theora можно заменить форматом WebM. Этот формат не будет работать на более старых версиях Firefox или Opera, но они не очень распространены в любом случае. Или же можно совсем разойтись и включить все три версии своего видео — H.264, Theora и WebM в указанном порядке. Версия WebM идет перед версией Theora для того, чтобы браузеры, которые поддерживают оба эти формата, выбрали видео лучшего качества.

Ну а если гулять по полной программе, то можно создать одну веб-страницу с видео как для настольных компьютеров, так и для мобильных устройств. В таком случае нужно не только предоставить файлы в формате H.264 и Theora, но также создать версии видеофайлов меньшего объема, более подходящие для менее мощных мобильных устройств и интернет-подключений с меньшей пропускной способностью. Чтобы обеспечить получение мобильными устройствами менее объемистых файлов, а настольными — файлов лучшего качества, нужно использовать технологию *запросов о возможностях воспроизведения устройства* (media queries), как рассматривается *во врезке "Малоизвестная или недооцененная возможность. Запросы о возможностях воспроизведения видео" главы 8*.

Резервное решение Flash

Испокон веков все браузеры обрабатывают нераспознаваемые теги одинаково — игнорируют их. Например, если Internet Explorer 8 встречается открывающий тег элемента `<video>`, он с ветерком проносится мимо него, не затрудняясь ознакомиться с атрибутом `src` и другими параметрами этого элемента. Но при всем этом, браузеры не игнорируют *содержимое* внутри неизвестного им элемента, что является важной особенностью. Это означает, что в случае следующей разметки:

```
<video controls width="400" height="300">
  <source src="discoParty.mp4" type="video/mp4">
  <source src="discoParty.ogv" type="video/ogg">
```

```
  <p>We like disco dancing.</p>
</video>
```

браузеры, которые не понимают HTML5, ведут себя, как будто бы они видели вот эту разметку:

```
<p>We like disco dancing.</p>
```

Эта особенность и лежит в основе бесшовного предоставления резервного решения для старых браузеров.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Кодирование файлов мультимедиа

Итак, вы уже знаете, какие комбинации форматов использовать, но не обязательно осведомлены, как преобразовать свои мультимедийные файлы в эти форматы. Не следует впадать из-за этого в отчаяние, т. к. существует множество инструментов для решения этой задачи. Некоторые обрабатывают сразу несколько файлов в пакете, другие имеют репутацию предоставления результатов профессионального качества (и соответствующую цену), а третьи работают на мощных веб-серверах, предоставляя немедленные результаты. Задача состоит в том, чтобы разобраться со всем этим разнообразием и выбрать кодировщик, отвечающий вашим требованиям и возможностям. Далее приводится список и краткое описание некоторых кодировщиков.

- **Аудиоредакторы.** Для преобразования файлов WAV в формат MP3 или Vorbis можно воспользоваться каким-либо базовым аудиоредактором. Audacity — один из таких редакторов для Mac и Windows, можно скачать бесплатно по адресу **http://audacity.sourceforge.net**. Но для поддержки этим редактором MP3 также нужно установить кодировщик LAME MP3, который можно загрузить по адресу **http://lame.buanzo.com.ar**. Редактор Goldwave (**www.goldwave.com**) предоставляет подобные возможности. Попробовать этот редактор можно бесплатно, а приобрести за номинальную плату.
- **Miro Video Converter.** Бесплатный редактор с открытым кодом для Windows и Mac OS X. Преобразует видеофайлы практически любого формата в формат WebM, Theora или H.264. Также имеет предустановленные настройки размеров экрана и поддерживаемых форматов для мобильных устройств, таких как iPad, iPhone и телефоны Android. Общим недостатком является отсутствие возможности тонкой настройки более продвинутых функций для управления процессом кодирования. Загрузить редактор можно по адресу **www.mirovideoconverter.com**.
- **Firefogg.** Этот модуль расширения для Firefox может создавать видеофайлы Theora и WebM, предоставляя больше опций, чем Miro Video Converter. Работает непосредственно в браузере, но вся работа выполняется локально без обращения к веб-серверу. Установить редактор можно по адресу **http://firefogg.org**.
- **HandBrake.** Эта многоплатформная программа с открытым исходным кодом преобразует широкий диапазон видеоформатов в формат H.264, а также пару других современных форматов. Доступна по адресу **http://handbrake.fr**.
- **Zencoder.** Профессиональный сервис для кодирования мультимедиа, который можно интегрировать в свой веб-сайт. Редактор загружает видеофайлы с веб-сервера, кодирует их в требуемый формат и с указанной частотой дискретизации присваивает им указанные названия, а затем размещает их в требуемом месте. Крупному сайту, такому как, например, сайт для обмена видеофайлами, пришлось бы платить этому сервису порядочный месячный сбор. Адрес сервиса — **http://zencoder.com**.

Примечание

Браузеры, поддерживающие HTML5-аудио, игнорируют резервное решение, даже если они не могут проигрывать основной файл. Например, когда Firefox сталкивается с элементом <video>, который содержит только файл формата H.264, но без альтернативного файла формата Theora, он выводит окно видеопроигрывателя, в котором отображается значок крестика (см. рис. 5.3), но не показывает резервное содержимое.

Теперь, после того как мы научились вставлять резервное содержимое, надо решить, какое именно содержимое вставлять. Одним из примеров плохого резервного содержимого будет текстовое сообщение, наподобие "Ваш браузер не поддерживает HTML5-видео, поэтому вам нужно обновить его". Посетители веб-сайтов считают сообщения такого типа крайне невежливыми и, скорее всего, не возвратятся на сайт, приветствующий их таким образом.

Правильный подход — это включить в качестве резервного содержимого другое работоспособное видеоокно, иными словами, любое содержимое, которое бы использовалось на обычной видеостранице, т. е. странице без поддержки HTTP5. Одной из возможностей будет окно YouTube. При использовании этого подхода нужно следовать правилам YouTube — протяженность видео должна быть меньше 15 минут и в нем не должно быть оскорбительного содержимого или содержимого, защищенного авторским правом. Видеофайл в лучшем формате загружается на YouTube, где он кодируется в форматы, поддерживаемые этим сервисом. Для реализации этого подхода перейдите на страницу **http://upload.youtube.com/my_videos_upload**.

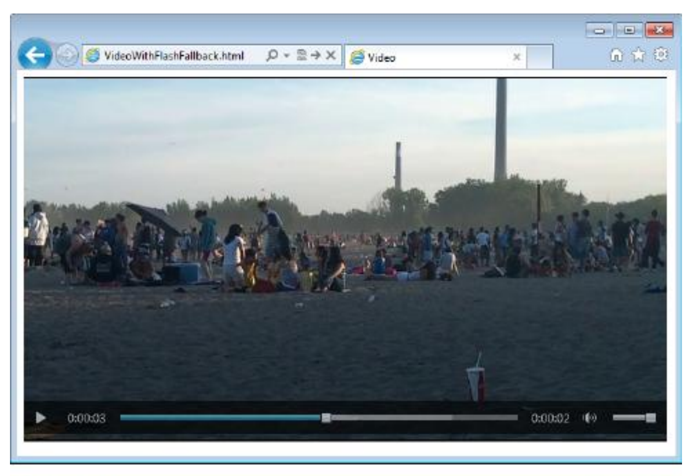
Другим подходом будет использовать видеопроигрыватель Flash (или аудиопроигрыватель Flash для аудио). К счастью, в Интернете существует масса видеопроигрывателей Flash, многие из которых бесплатные, по крайней мере, для некоммерческого использования. И большинство из них поддерживает формат H.264, который вы уже, наверное, используете для HTML5-видео.

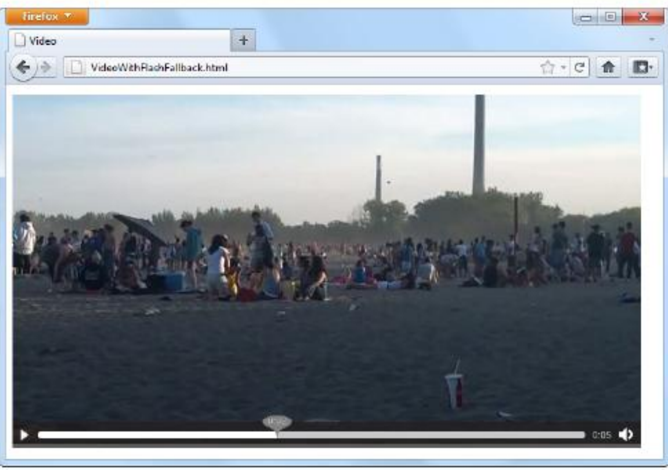
В следующем листинге приведен пример использования в качестве резервного решения в элементе <video> проигрывателя Flowplayer (**http://flowplayer.org**):

```
<video controls width="700" height="400">
  <source src="beach.mp4" type="video/mp4">
  <source src="beach.ogv" type="video/ogg">

  <object id="flowplayer" width="700" height="400"
          data="flowplayer-3.2.7.swf"
          type="application/x-shockwave-flash">
    <param name="movie" value="flowplayer-3.2.7.swf">
    <param name="flashvars" value='config={"clip":"beach.mp4"}'>
  </object>
</video>
```

Жирным шрифтом в третьей снизу строке кода выделено имя файла, который браузер передает в качестве параметра видеопроигрывателю Flowplayer. Хотя для этого примера возможны три результата — HTML5-видео с H.264, HTML5-видео





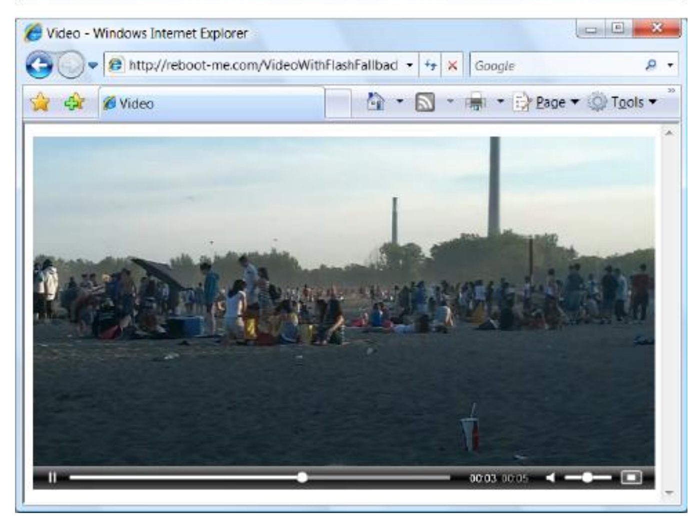


Рис. 5.5. Одно и то же видео отображено тремя разными способами: в Internet Explorer 9 *(вверху)*, в Firefox *(в центре)* и в Internet Explorer 7 *(внизу)*

с Theora или Flash-видео с H.264, для него требуются только два видеофайла, что позволяет сэкономить на работе по кодированию. Результаты исполнения примера показаны на рис. 5.5.

Совет

Хотя использование файла H.264 с Flash и удобно, этот подход не совсем идеальный. Поддержка этого формата была введена только в версии Flash 9.0.115.0. Браузеры, на которых установлен более старый модуль Flash, не смогут проигрывать видеофайл в формате H.264 без обновления. Можно было бы использовать формат Flash Video Format (flv), который будет гарантированно работать на всех версиях модуля Flash, но для этого потребуется выполнять еще один заход перекодировки.

Но опять же вероятна ситуация, что некоторые посетители вашего веб-сайта пользуются браузером, на котором не установлен модуль Flash и который не поддерживает HTML5. Для таких пользователей можно реализовать еще одно резервное решение, например, в виде ссылки для загрузки видеофайла, который потом можно будет просмотреть в соответствующем проигрывателе. Это резервное решение вставляется после решения Flash, но все еще внутри элемента `<object>` следующим образом:

```
<video controls width="700" height="400">
  <source src="beach.mp4" type="video/mp4">
  <source src="beach.ogv" type="video/ogg">

  <object id="flowplayer" width="700" height="400"
          data="http://releases.flowplayer.org/swf/flowplayer-3.2.7.swf"
          type="application/x-shockwave-flash">
    <param name="movie" value="beach.mp4">

    <img src="beach_thumbnail.jpg" alt="A lazy day at the beach">
    <p>Your browser does not support HTML5 video or Flash.</p>
    <p>You can download the video in <a href="beach.mp4">MP4 H.264</a>
       or <a href="beach.ogv">Ogg Theora</a> format.</p>
  </object>
</video>
```

Если же требуется, наоборот, реализовать основное решение в виде Flash, а резервное — в виде HTML, нужно просто переставить строки из предыдущего примера. Начинаем с элемента `<object>`, в который вставляем элемент `<video>` непосредственно перед закрытием тега `</object>`. А содержимое самого последнего резервного решения вставляется после последнего элемента `<source>`:

```
<object id="flowplayer" width="700" height="400"
        data="http://releases.flowplayer.org/swf/flowplayer-3.2.7.swf"
        type="application/x-shockwave-flash">
  <param name="movie" value="butterfly.mp4">

  <video controls width="700" height="400">
    <source src="beach.mp4" type="video/mp4">
    <source src="beach.ogv" type="video/ogg">
```

```
    <img src="beach_thumbnail.jpg" alt="A lazy day at the beach">
    <p>Your browser does not support HTML5 video or Flash.</p>
    <p>You can download the video in <a href="beach.mp4">MP4 H.264</a>
       or <a href="beach.ogv">Ogg Theora</a> format.</p>
  </video>
</object>
```

Обычно этот подход следует применять только в том случае, если нужно расширить существующий веб-сайт на основе Flash для поддержки устройств Apple, таких как iPad. Кстати, существует по крайней мере один проигрыватель на JavaScript со встроенной возможностью резервного решения HTML5. Называется он JW Player, а загрузить его можно по адресу **www.longtailvideo.com/players/jw-flv-player**.

Управление плеером посредством JavaScript

На данный момент мы изучили довольно сложный материал. Мы теперь знаем, как с помощью элементов <audio> и <video> создать решение с достаточно удовлетворительным уровнем браузерной поддержки, которое работает на большем количестве браузеров, чем сегодняшние решения на основе Flash. Совсем неплохо для необкатанной новой технологии!

Используя только HTML-разметку, это, пожалуй, все, что можно выжать из элементов <audio> и <video>. Но для обоих элементов существует обширная объектная модель JavaScript, посредством которой можно управлять воспроизведением с помощью сценариев. Более того, используя этот подход, можно даже настроить некоторые детали, такие как, например, скорость воспроизведения, что невозможно со стандартными аудио- и видеопроигрывателями.

В следующих разделах мы изучим поддержку JavaScript, рассмотрев два практических примера. Прежде всего, мы добавим звуковые эффекты в игру, а потом создадим специализированный видеопроигрыватель. В завершение мы рассмотрим решения, созданные другими разработчиками с использованием мощной смеси HTML5 и JavaScript, включая высокопроизводительные проигрыватели со скинами и редактируемыми субтитрами.

Добавление звуковых эффектов

Элемент <audio> предоставляет более обширные возможности, чем простое проигрывание песен и других звукозаписей посетителями веб-страницы. С его помощью можно также проигрывать звуковые эффекты в любое требуемое время. Эта возможность делает его особенно полезным в тех случаях, когда к игре требуется добавить музыку и звуковые эффекты.

На рис. 5.6 показана простая игра, которую можно усовершенствовать с помощью подобной возможности.

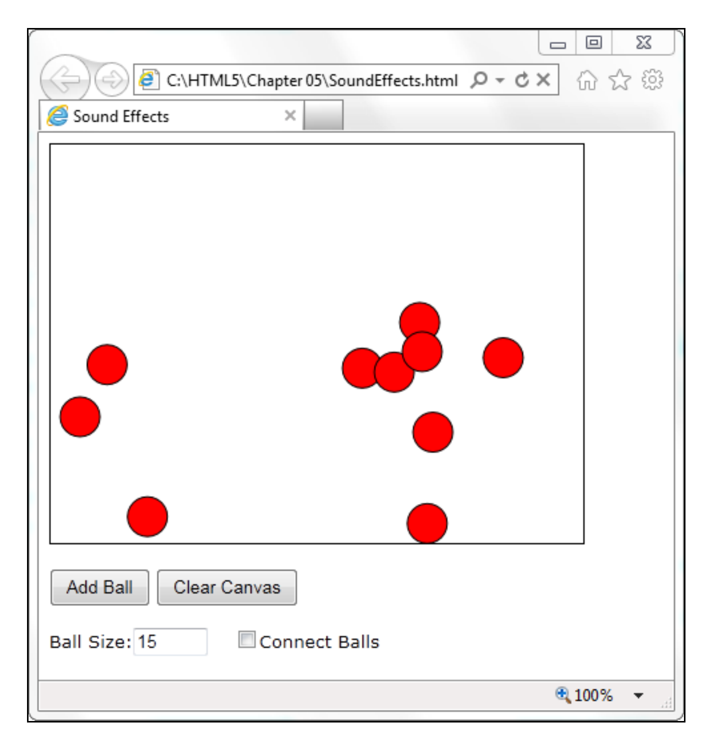


Рис. 5.6. На этой веб-странице выполняется простая анимация, состоящая из стилизованных мячиков, отскакивающих от нижнего и боковых краев рамки (пола и стен). Посетитель может добавить новый мячик, щелкнув кнопкой мыши по полю игры, или, щелкнув по мячику, направить его в другом направлении

Полностью код примера рассматривается в *главе 6* при обсуждении элемента `<canvas>`. На данном же этапе мы только рассмотрим, как реализовать соответствующий звуковой фон для этой анимации.

В этом примере сочетаются звуковая дорожка фонового сопровождения и звуковые эффекты. Реализация фонового сопровождения — это самая легкая часть стоящей перед нами задачи. Первым делом вставляем в разметку невидимый элемент `<audio>`:

```
<audio id="backgroundMusic" loop>
  <source src="TheOwlNamedOrion.mp3" type="audio/mp3">
  <source src="TheOwlNamedOrion.ogg" type="audio/ogg">
</audio>
```

Так как для этого проигрывателя не указаны атрибуты `autoplay` и `controls`, то после запуска его не видно и не слышно. А указанный атрибут `loop` означает, что когда воспроизведение дорожки начнется, она будет продолжаться бесконечно. Для управления воспроизведением нам требуются два метода аудиообъекта (или видеообъекта): `play()` и `pause()`. Несколько неожиданно, но метода `stop()` не существует, и для остановки воспроизведения нужно сначала его приостановить методом `pause()`, а потом сбросить значение свойства `currentTime` в `0`, что представляет начало файла.

Приняв все это во внимание, не составляет труда начать воспроизведение фоновой дорожки при создании первого мячика:

```
var audioElement = document.getElementById("backgroundMusic");
audioElement.play();
```

И так же легко остановить воспроизведение при очистке холста:

```
var audioElement = document.getElementById("backgroundMusic");
audioElement.pause();
audioElement.currentTime = 0;
```

Как упоминалось ранее, можно запустить неограниченное количество проигрывателей `<audio>` одновременно. Поэтому, запустив воспроизведение фонового сопровождения, мы можем приступить к более сложной задаче добавления звуковых эффектов.

В данном примере, всякий раз, когда мячик отскакивает от "стены" или "пола", проигрывается звуковой эффект "боинк"[1]. Для разнообразия слышны несколько слегка отличающиеся друг от друга звуки "боинк". Это подобие более реалистической игры, в которой, как правило, применяется десяток или больше разных звуков.

Реализовать этот замысел можно несколькими способами, но не все они будут практичными. Как вариант, можно создать один элемент `<audio>` для проигрывания всех звуковых эффектов. Потом при каждом столкновении в этот элемент можно загружать один из аудиофайлов (посредством установки значения атрибута `src`) и проигрывать его. Но этот подход наталкивается на две преграды. Первая: один элемент `<audio>` может одновременно воспроизводить только один звук. Таким образом, если несколько мячиков столкнутся со стеной или полом одновременно или в быстрой последовательности, нужно будет либо игнорировать второй звук, перекрывающий первый, либо остановить воспроизведение первого звука, чтобы начать проигрывать второй. Другая проблема состоит в том, что установка значения свойства `src` заставляет браузер запрашивать аудиофайл. И если некоторые браузеры справляются с этой задачей достаточно быстро (когда файл уже находится в кэше), то Internet Explorer — нет. В результате получаем запоздалое аудио, иными словами, "боинк" звучит полсекунды после столкновения.

Лучше использовать группу элементов `<audio>`, по одному для каждого звука. Далее показан пример кода для создания такой группы:

```
<audio id="audio1">
  <source src="boing1.mp3" type="audio/mp3">
  <source src="boing1.wav" type="audio/wav">
</audio>
<audio id="audio2">
  <source src="boing2.mp3" type="audio/mp3">
  <source src="boing2.wav" type="audio/wav">
</audio>
```

[1] Этот звук похож на звук от вибрирующей пружины или от варгана — язычкового народного инструмента. — *Ред.*

```
<audio id="audio3">
  <source src="boing3.mp3" type="audio/mp3">
  <source src="boing3.wav" type="audio/wav">
</audio>
```

ПРИМЕЧАНИЕ

Не обязательно, чтобы все три элемента `<audio>` проигрывали разные аудиофайлы. Например, если использовать только один звук, но обеспечить возможность перекрывающихся звуков от столкновений нескольких мячиков, все равно следует применять три аудиопроигрывателя.

Когда мячик сталкивается с преградой, код JavaScript вызывает специальную функцию `boing()`. Эта функция берет очередной элемент `<audio>` и проигрывает его.

Реализуется все это следующим кодом:

```
// Отслеживаем количество элементов <audio>.
var audioElementCount = 3;

// Отслеживаем элемент <audio>, следующий в очереди на проигрывание.
var audioElementIndex = 1;

function boing() {
  // Берем следующий в очереди на проигрывание элемент <audio>.
  var audioElementName = "audio" + audioElementIndex;
  var audio = document.getElementById(audioElementName);

  // Воспроизводим звуковой эффект.
  audio.currentTime = 0;
  audio.play();

  // Переводим счетчик на следующий элемент <audio>.
  if (audioElementIndex == audioElementCount) {
      audioElementIndex = 1;
  }
  else {
      audioElementIndex += 1;
  }
}
```

СОВЕТ

Ознакомиться с работой этого примера и его звуковыми эффектами можно на сайте книги **http://www.prosetech.com/html5/**.

Этот подход работает удовлетворительно. Но если требуется использовать намного больший диапазон звуковых эффектов? Реализовать это будет легче всего, создав скрытый элемент `<audio>` для каждого звука. Если по каким-либо причинам это невозможно, тогда следует динамически устанавливать значение свойства атрибута

`src` имеющегося элемента `<audio>`. Можно также динамически создавать новый элемент `<audio>`, как показано в следующем коде:

```
var audio = document.createElement("audio");
audio.src = "newsound.mp3";
```

или использовать этот быстрый вызов:

```
var audio = new Audio("newsound.mp3");
```

Но оба эти подхода имеют потенциальные проблемы. Первая проблема: аудиофайл нужно указать задолго до его воспроизведения. В противном случае воспроизведение будет запоздалым, особенно в Internet Explorer. Вторая проблема: нужно знать, какие аудиоформаты поддерживаются, чтобы можно было установить правильный тип файла. Это требует применения неуклюжего метода `canPlayType()`. Методу передается MIME-тип аудио или видео, и он определяет, может ли браузер воспроизводить этот формат. Впрочем, сказать "определяет" будет не совсем верно. В действительности метод `canPlayType()` возвращает: пустую строку, если данный формат не поддерживается браузером; строку `probably`, если он думает, что поддерживается; и строку `maybe`, если он надеется, что, возможно, поддерживается, но не может давать никаких обещаний. Такая довольно непривлекательная ситуация существует вследствие того, что поддерживаемые форматы контейнера могут использовать неподдерживаемые кодеки, а поддерживаемые кодеки могут использовать неподдерживаемые настройки кодирования.

Большинство разработчиков останавливается на коде, наподобие показанного в листинге далее, который пытается воспроизводить файл, если метод `canPlayType()` возвращает любой ответ, кроме пустой строки:

```
if (audio.canPlayType("audio/ogg")) {
    audio.src = "newsound.ogg";
}
else if (audio.canPlayType("audio/mp3")) {
  audio.src = "newsound.mp3";
}
```

Создание своего видеопроигрывателя

Одним из основных поводов заняться углубленным изучением программирования элементов `<audio>` и `<video>` в JavaScript будет создание собственного проигрывателя. Основная идея заключается в простоте процесса — удаляем атрибут `controls`, чтобы было только окно воспроизведения, и добавляем внизу его свои кнопки управления воспроизведением. А чтобы эти кнопки функционировали, добавляем соответствующий JavaScript-код. Пример полученного таким образом видеопроигрывателя показан на рис. 5.7.

Любому видеопроигрывателю требуется базовый набор кнопок управления воспроизведением. Стандартные кнопки управления воспроизведением проигрывателя на рис. 5.7 создаются следующим кодом:

```
<button onclick="play()">Play</button>
<button onclick="pause()">Pause</button>
<button onclick="stop()">Stop</button>
```

Нажатие этих кнопок активирует следующие сверхпростые функции:

```
function play() {
  video.play();
}
function pause() {
  video.pause();
}
function stop() {
  video.pause();
  video.currentTime = 0;
}
```

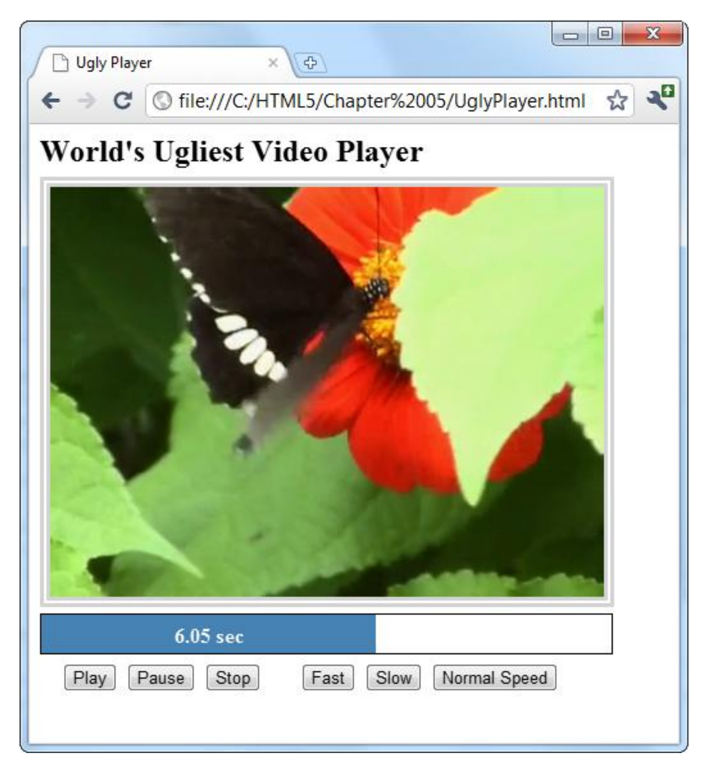


Рис. 5.7. Создание своего видеопроигрывателя HTML5 не составляет никакого труда (чего нельзя сказать о придании ему привлекательного внешнего вида). Данный проигрыватель оснащен стандартными кнопками управления воспроизведением, индикатором хода воспроизведения, а также несколькими дополнительными кнопками, которые демонстрируют, что можно делать с элементом `<video>` с помощью JavaScript

Функции других кнопок управления воспроизведением не совсем стандартные — они используются для регулирования скорости воспроизведения путем значения свойства `playbackRate`. Например, при значении `playbackRate` равным `2` видео про-

игрывается вдвое быстрее нормальной скорости, но с откорректированной высотой тона, вследствие чего звук воспроизводится нормально, только вдвое быстрее. Это замечательная возможность для ускоренного просмотра медленных видеоинструкций. Аналогично, при значении `playbackRate` равном `0.5` видео проигрывается со скоростью, составляющей половину нормальной скорости. При значении `playbackRate` равном `-1` видео должно проигрываться с нормальной скоростью, только в обратном направлении, но браузеры сталкиваются с проблемой реализации этого режима должным образом. Код для реализации этих функций следующий:

```
function speedUp() {
  video.play();
  video.playbackRate = 2;
}

function slowDown() {
  video.play();
  video.playbackRate = 0.5;
}

function normalSpeed() {
  video.play();
  video.playbackRate = 1;
}
```

Задача создания индикатора хода воспроизведения представляет несколько больший интерес. В отношении разметки он создается из двух элементов `<div>`, один из которых вложен в другой:

```
<div id="durationBar">
  <div id="positionBar"><span id="displayStatus">Idle.</span></div>
</div>
```

Совет

Индикатор хода воспроизведения является примером ситуации, идеально подходящей для использования элемента `<progress>` *(см. разд. "Индикатор выполнения <progress> и счетчик <meter>" главы 4)*. Но уровень браузерной поддержки элемента `<progress>` все еще низкий, намного ниже, чем возможности видео HTML5. Поэтому в этом примере подобно выглядящий индикатор создается с помощью двух элементов `<div>`.

Внешний элемент `<div>` (с именем `durationBar`) рисует черную рамку, которая обрамляет весь индикатор и имитирует полную продолжительность видео. Внутренний элемент `<div>` (с именем `positionBar`) указывает текущую точку воспроизведения, заполняя часть черного индикатора синим цветом. Наконец, элемент `<span>` внутри внутреннего элемента `<div>` содержит текст, указывающий текущую позицию воспроизведения в секундах.

Далее приведены правила таблицы стилей, которые устанавливают размер индикаторов и окрашивают их в соответствующие цвета:

```
#durationBar {
  border:        solid 1px black;
  width:         100%;
  margin-bottom: 5px;
}

#positionBar {
  height:      30px;
  color:       white;
  font-weight: bold;
  background:  steelblue;
  text-align:  center;
}
```

В процессе воспроизведения элемент `<video>` постоянно активирует событие `onTimeUpdate`. Реагируя на это событие, обновляем индикатор хода воспроизведения:

```
<video id="videoPlayer" ontimeupdate="progressUpdate()">
  <source src="butterfly.mp4" type="video/mp4">
  <source src="butterfly.ogv" type="video/ogg">
</video>
```

А этот код получает от свойства `currentTime` значение текущей позиции в видео, разделяет его на общее время (свойство `duration`) и преобразует результат в процентное значение, которое используется для установления размера `<div>`-элемента `postitionBar`:

```
function progressUpdate() {
  // Корректируем длину синего <div>-элемента positionBar, от 0 к 100%.
  var positionBar = document.getElementById("positionBar");
  positionBar.style.width = (video.currentTime /
                             video.duration * 100) + "%";
  // Выводим число секунд до двух десятичных разрядов.
  displayStatus.innerHTML = (Math.round(video.currentTime*100)/100) +
                             " sec";
}
```

Совет

Можно сделать индикатор выполнения немного замысловатей, добавив индикатор загрузки, показывающий объем содержимого, загруженного и помещенного в буфер в данный момент. Эта возможность уже имеется в проигрывателях, встроенных в браузеры. Чтобы добавить этот индикатор, нужно обрабатывать событие `onProgress` и работать со свойством `seekable`. Дополнительную информацию о свойствах, методах и событиях элемента `<video>` см. в справочных материалах Microsoft по адресу **http://msdn.microsoft.com/ru-ru/library/ff975073.aspx**.

Проигрыватели на JavaScript

Если вы не хотите ни от кого зависеть в своей работе, то можете создать свой аудио- или видеопроигрыватель с чистого листа. Но это будет очень трудоемким проектом, особенно если вы хотите обеспечить разные вычурные возможности, наподобие интерактивного списка воспроизведения. К тому же, если за вашей спиной нет небольшого дизайнерского отдела, то существует отчетливая вероятность, что конечный продукт ваших усилий будет иметь слегка уродливый внешний вид.

К счастью, для веб-дизайнеров, блуждающих в поисках идеального HTML5-проигрывателя, имеется лучший вариант. Вместо того чтобы разрабатывать мультимедийный проигрыватель самому, можно взять бесплатный, настраиваемый посредством JavaScript проигрыватель в Интернете. Два из них — это VideoJS (**http://videojs.com**) и, для фанатов библиотеки jQuery, jPlayer (**www.jplayer.org**). Оба проигрывателя легковесные, удобны в использовании, а также имеют функцию смены скинов с помощью таблицы стилей (рис. 5.8).



Рис. 5.8. Проигрыватель VideoJS имеет функцию смены скинов, которые имитируют внешний вид популярных видеовеб-сайтов, включая YouTube, Vimeo и Hulu.
Здесь показано изменение элементов управления воспроизведением при смене скинов

Большинство проигрывателей мультимедиа на JavaScript (включая проигрыватели VideoJS и jPlayer) содержат встроенные резервные решения Flash, что позволяет разработчику сэкономить время и усилия на поиски отдельного Flash-проигрывателя. А проигрыватель jPlayer имеет удобную функцию создания списка воспроизведения, позволяющего выбрать для проигрывания или просмотра несколько файлов (рис. 5.9).

Чтобы использовать VideoJS в своих разработках, сначала загрузите его JavaScript-файлы с веб-сайта проигрывателя. Потом вставьте ссылку на его сценарий и таблицы стилей, как показано в следующем коде:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>...</title>

    <script src="video.js"></script>
    <link rel="stylesheet" href="video-js.css">
  </head>
  ...
```

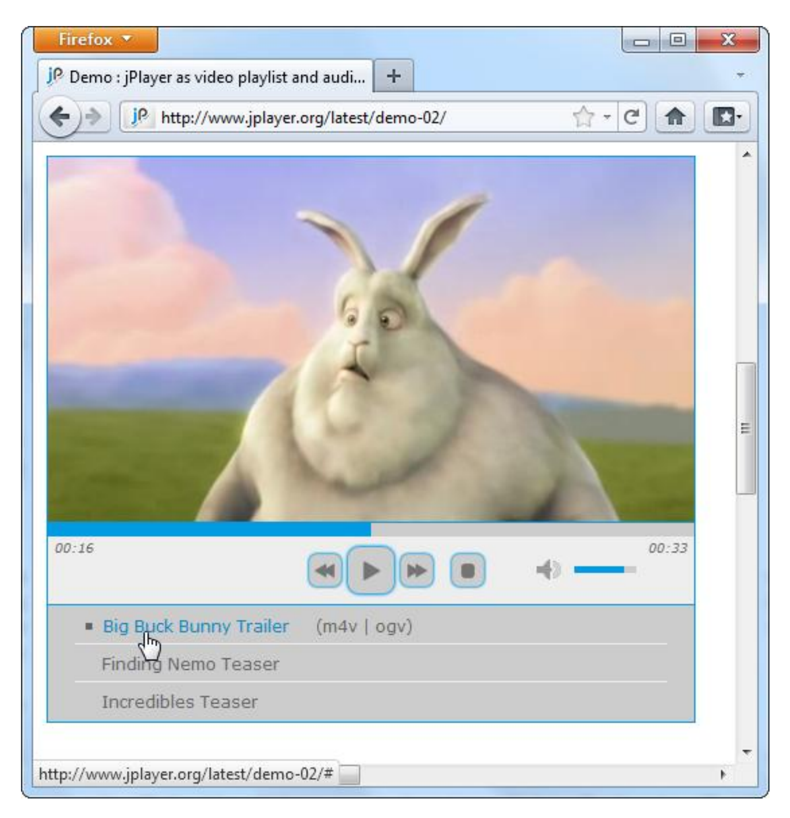


Рис. 5.9. Функция создания списка воспроизведения проигрывателя jPlayer позволяет поставить в очередь для проигрывания несколько файлов. Файлы в списке можно проигрывать в существующей последовательности или выбрать требуемый файл, щелкнув на нем мышью. Здесь список содержит три видеофайла

Затем используется обычный элемент <video> с несколькими элементами <source> и резервным решением Flash. (Для резервного Flash-решения в проигрывателе VideoJS используется Flowplayer, но его можно удалить и использовать другой Flash-проигрыватель.) По сути, единственная разница между обычной страницей HTML5 и страницей с проигрывателем VideoJS состоит в том, что в последней нужно использовать специальный элемент <div>, чтобы вставить в него видеопроигрыватель:

```
<div class="video-js-box">
   <video class="video-js" width="640" height="264" controls ...>
     ...
   </video>
</div>
```

Приятно видеть, что даже при расширении HTML5 работа с ним остается довольно легкой.

Субтитры и доступность

Как мы узнали в предыдущих главах, создатели HTML5 часто уделяли внимание вопросу доступности веб-страниц, иными словами, люди с ограниченными возможностями легко и эффективно могут пользоваться расширенными веб-страницами.

Добавление информации в страницу для повышения уровня ее доступности — довольно простая задача. Нужно просто добавить соответствующее описание в атрибуте `alt`. Но что может быть эквивалентом описания `<alt>` для видеопотока? Общим мнением по этому вопросу является использование *субтитров*, т. е. текстовых надписей, которые выводятся в соответствующем месте при воспроизведении. Субтитры могут быть подобны телевизионным субтитрам, т. е. просто транскрипцией диалога, или же описательными и предоставлять дополнительную информацию. Суть заключается в том, что они предоставляют пользователю возможность следить за происходящим в видео, даже если у человека проблемы со слухом (или если он не хочет, чтобы весь офис знал, что он смотрит боевик, вместо того, чтобы работать над отчетом).

К сожалению, несмотря на то, что использование субтитров явно выделяется как идеальное решение, согласие о том, как именно его реализовывать, никак не может быть достигнуто. В HTML5 предлагается использовать элемент `<track>`, который будет указывать на связанный файл субтитров, возможно, используя формат WebSRT или более новый и все еще развивающий формат WebVTT. В ближайшем будущем браузеры смогут считывать этот элемент и предоставлять пользователям элементы управления для включения и выключения субтитров, а поисковые системы смогут извлекать файлы субтитров и индексировать их содержимое. Но в настоящее время окончательные подробности еще не согласованы, и никакие официальные действия не предпринимались.

Но разработчики уже создали собственные обходные решения на JavaScript для предоставления субтитров. Обычно в этих решениях используется элемент `<span>`, который выводится поверх окна видео и заполняется текстом субтитров в соответствующих местах воспроизведения. Например, на веб-сайте **www.storiesinflight.com/js_videosub** можно загрузить JavaScript-библиотеку VideoSub, с помощью которой можно извлекать текст из файла субтитров в формате WebSRT и отображать его в окне видео (рис. 5.10).

Сценарий VideoSub проверяет поддержку субтитров браузером и вступает в действие, только если браузер не имеет этой возможности. (В настоящее время, это

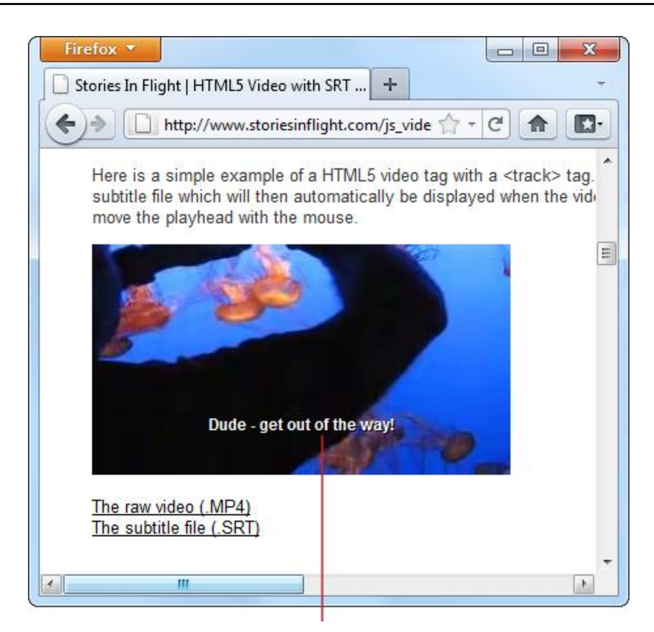


Рис. 5.10. Две ссылки на сценарий JavaScript и файл WebSRT — вот и все, что потребовалось, чтобы добавить субтитры к этому видео

всегда, т. к. ни один из браузеров не поддерживает субтитров.) Существует даже полноценный проигрыватель на JavaScript — LeanBack Player, со встроенной поддержкой субтитров (**http://dev.mennerich.name/showroom/html5_video**). К сожалению, еще слишком рано назвать наилучший подход к реализации субтитров, подход, который будет работать с минимальными усилиями на браузерах следующего поколения. Поэтому, если вам действительно нужны субтитры сейчас, вам следует продолжать использовать JavaScript и быть готовым к модифицированию своих страниц по мере развития HTML.

6

ГЛАВА 6

Основы рисования на холсте

Как мы узнали в *главе 1*, одна из целей HTML5 — облегчить внедрение расширенных приложений в обычные во всех других отношениях веб-страницы. Термин "расширенное приложение" означает не приложение большого объема, а приложение с расширенными возможностями, такими как искусная графика, интерактивные функции и анимация.

Самым важным новым инструментом для расширенных приложений является *холст* — поверхность для рисования, на которой пользователь может дать волю своим непризнанным художественным способностям. Холст стоит обособленно от всех других элементов HTML, т. к. для работы с ним *требуется* JavaScript. Иного способа для черчения фигур или рисования изображений попросту нет. Это означает, что холст, по сути, является средством программирования, таким, которое позволяет выйти далеко за пределы первоначального концепта Интернета, согласно которому он основывается на содержимом документного типа.

С первого взгляда использование холста может казаться похожим на использование программы MS Paint, вставленной в окно браузера. Но, копнув поглубже, можно увидеть, что холст — ключевой компонент для ряда графически продвинутых приложений, включая некоторые приложения, о которых вы уже, наверное, и сами подумали — игры, картографические инструменты и динамические графики, и такие, которые вы, возможно, не могли вообразить — музыкально-световые представления и эмуляторы физических процессов. В не очень далеком прошлом создание таких приложений без помощи модулей расширения, таких как Flash, являлось чрезвычайно сложной задачей. Сегодня же холст внезапно делает все эти приложения возможными, если, конечно, вы готовы на дополнительные временные и интеллектуальные затраты.

В этой главе мы узнаем, как создавать холст и рисовать на нем прямые и кривые линии и простые фигуры. Потом мы дадим нашим новоприобретенным навыкам практическое применение, создав простую программу рисования. Также, возможно, самое главное, мы узнаем *(см. разд. "Холст на заполнителе" далее в этой главе)*, как заставить страницы с холстом работать на старых браузерах, которые не поддерживают HTML5.

ПРИМЕЧАНИЕ

Для одних разработчиков холст будет незаменим, в то время как для других он будет всего лишь интересным развлечением. (А для третьих он, возможно, будет представлять интерес, но требовать слишком много усилий по сравнению с отработанной платформой программирования наподобие Flash.) Но можно быть уверенным в одном: этой простой поверхности для рисования отводится намного более важная роль, чем простой забавы для развлечения скучающих программистов.

Базовые возможности холста

Элемент `<canvas>` предоставляет рабочее пространство для рисования. С точки зрения разметки, это простой до предела элемент с тремя атрибутами — `id`, `width` и `height`:

```
<canvas id="drawingCanvas" width="500" height="300"></canvas>
```

Атрибут `id` присваивает данному холсту однозначное имя, требуемое для его идентификации кодом JavaScript. А атрибуты `width` и `height` устанавливают ширину и высоту холста в пикселах, соответственно.

ПРИМЕЧАНИЕ

Размеры холста всегда следует устанавливать посредством атрибутов `width` и `height` элемента `<canvas>`, а не с помощью свойств `width` и `height` таблицы стилей. В противном случае возможно возникновение проблемы, которая рассматривается *во врезке "Аварийная ситуация. Все мои рисунки искажены!" главы 7*.

Обычно холст отображается как пустой прямоугольник без рамки, т. е. он не виден вообще. Чтобы сделать холст видимым, с помощью таблицы стилей ему можно задать цветной фон или рамку, как показано в следующем коде:

```
canvas {
  border: 1px dashed black;
}
```

Полученный результат, который будет нашей отправной точкой в изучении холста, показан на рис. 6.1.

Чтобы рисовать на холсте, нужно написать определенный объем кода JavaScript. Эта задача состоит из двух этапов. Первым делом наш сценарий должен получить объект холста, для чего используется метод `document.getElementById()`:

```
var canvas = document.getElementById("drawingCanvas");
```

В этом нет ничего особенного, т. к. метод `getElementById()` применяется во всех случаях, когда необходимо найти элемент на текущей странице.

ПРИМЕЧАНИЕ

Если вы не в ладах с JavaScript, то не продвинетесь далеко в работе с холстом. Чтобы получить минимально необходимые знания JavaScript, ознакомьтесь с материалом в *приложении 2*. Чтобы получить более обширные знания в этой области, обратитесь к специальной литературе.

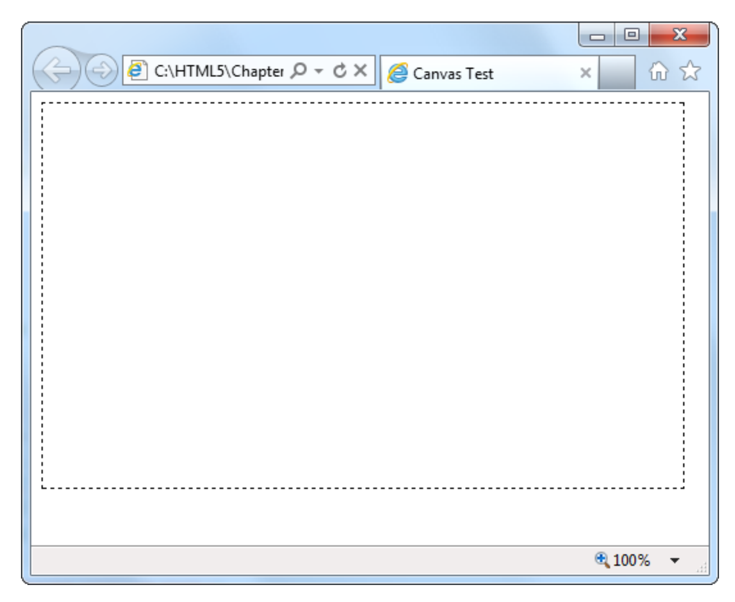


Рис. 6.1. Изначально холст — всего лишь пустой прямоугольник на веб-странице. Чтобы нарисовать на нем хотя бы одну линию, нужно написать несколько строчек кода JavaScript

Теперь надо получить двумерный контекст рисования, для чего применяется метод `getContext()`:

```
var context = canvas.getContext("2d");
```

Контекст можно рассматривать как сверхмощный инструмент рисования, который выполняет все необходимые для этого операции, такие как создание прямоугольников, печатание текста, вставка изображений и т. п. Это что-то наподобие универсальной мастерской для операций рисования на холсте.

Примечание

Тот факт, что контекст явно называется двумерным (и в коде указывается как "2d"), порождает очевидный вопрос, а именно: существует ли трехмерный контекст рисования? Ответ на этот вопрос — пока нет, но ясно, что создатели HTML5 оставили место для него в будущем.

Получить объект контекста и начать рисование можно в любой момент, например, сразу же после загрузки страницы, когда пользователь щелкнет мышью, и т. п. Вам, скорее всего, уже не терпится создать страницу, на которой можно было бы сразу же приступить к практической работе с холстом. В следующем листинге приведен код для создания шаблона такой страницы:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="ut-f-8">
  <title>Canvas Test</title>
```

```
  <style>
   canvas {
      border: 1px dashed black;
   }
  </style>

  <script>
   window.onload = function() {
      var canvas = document.getElementById("drawingCanvas");
      var context = canvas.getContext("2d");

      // (Код для рисования вставляется сюда)
   };
   </script>
</head>
<body>
  <canvas id="drawingCanvas" width="500" height="300"></canvas>
</body>
</html>
```

В разделе разметки `<style>` для холста создается рамка, указывающая его местонахождение на странице. А в разделе `<script>` обрабатывается событие `window.onload`, которое происходит после полной загрузки страницы браузером. После этого код получает объект холста и создает контекст рисования, подготовившись таким образом к рисованию. Эту разметку можно использовать в качестве отправной точки для дальнейших экспериментов с холстом.

> ***Примечание***
>
> Конечно же, при использовании холста в настоящей странице веб-сайта разметку следует немного разгрузить, удалив из нее код JavaScript и поместив его во внешний файл сценариев. Как это сделать, *см. в разд. "Перемещение кода JavaScript в файл сценариев" приложения 2*. Но на данном этапе удобнее, чтобы этот код находился в шаблоне страницы холста. Если вы не в настроении вводить код примера вручную, эту разметку можно взять из файла CanvasTemplate.html на сайте книги (**www.pmsetech.com/html5**).

Прямые линии

Теперь мы почти готовы приступить к рисованию. Почему почти? Потому что прежде нам нужно разобраться с системой координат холста. Принцип ее работы показан на рис. 6.2.

Самой простой фигурой, которую можно нарисовать на холсте, будет прямая линия. Для этого нужно выполнить три действия с контекстом. Сперва надо указать начальную точку линии с помощью метода `moveTo()`. Потом с помощью метода `lineTo()` задать конечную точку линии. Наконец, метод `stroke()` собственно рисует линию:

```
context.moveTo(10,10);
context.lineTo(400,40);
context.stroke();
```

С точки зрения обычного рисования эти действия можно рассматривать так: сначала мы ставим карандашом начальную точку линии (метод `moveTo()`), потом ставим конечную (метод `lineTo()`) и, наконец, собственно рисуем линию, соединяя ее начальную и конечную точки (метод `stroke()`). В результате исполнения этого кода точки (10, 10) и (400, 40) соединяются линией толщиной в один пиксел.

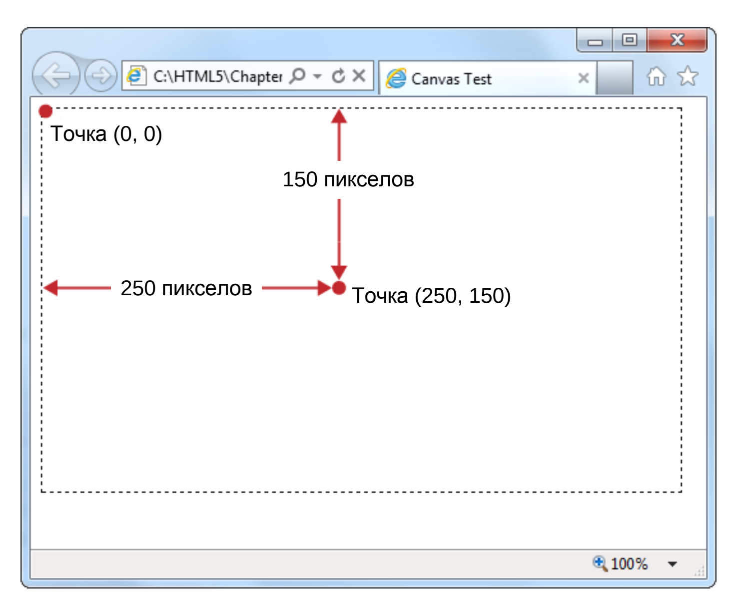


Рис. 6.2. Координаты левой верхней точки холста, называющейся *исходной точкой*, равны (0, 0). Значение абсциссы (т. е. *x*-координаты) увеличивается при перемещении вправо от исходной точки, а значение ординаты (т. е. *y*-координаты) — при перемещении вниз. Таким образом, для холста размером в 500×300 пикселов координаты конечной точки (правого нижнего угла) холста будут равны (500, 300)

К счастью, стиль линии можно разнообразить. На любом этапе рисования линии, но перед тем как вызывать метод `stroke()`, можно установить три свойства контекста рисования: `lineWidth`, `strokeStyle` и `lineCap`. Эти свойства продолжают действовать до тех пор, пока не будут изменены.

Свойство `lineWidth` определяет толщину линии в пикселах. Например, следующая строка кода устанавливает толщину линии в 10 пикселов:

```
context.lineWidth = 10;
```

Свойство `strokeStyle` определяет цвет линий. Значение этого свойства может быть в виде названия цвета HTML, кода цвета HTML или же CSS-функции `rgb()`, которая позволяет создать цвет из его красной, зеленой и синей составляющей. (Это полезно, т. к. большинство программ черчения и рисования используют такую же

систему.) Независимо от выбранного вами подхода, значения цвета нужно заключить в кавычки, как показано ниже:

```
// Устанавливаем красно-коричневый цвет посредством кода цвета HTML:
context.strokeStyle = "#cd2S2S";

// Устанавливаем красно-коричневый цвет посредством функции rgb():
function: context.strokeStyle = "rgb(205,40,40)";
```

> ***Примечание***
>
> Свойство `strokeStyle` называется именно так, а не `strokeColor`, т. к. позволяет устанавливать не только чистые цвета. Как мы увидим далее, цвет можно задавать в виде градиентной закраски *(см. разд. "Градиентная заливка фигур" главы 7)* и в виде узора из изображений (*см. разд. "Заполнение фигур изображениями" главы 7)*.

Наконец, свойство `lineCap` указывает тип концов линии. По умолчанию этому свойству присваивается значение `butt` (что придает концам линии прямоугольную форму), но можно также присвоить значение `round`, делая концы округлыми, и `squre`. Последнее значение также делает концы линии прямоугольными, как и значение `butt`, но удлиняет ее на каждом конце на половину значения толщины линии. Таким же образом линия удлиняется и при форме концов `round`.

Далее приведен код для рисования трех горизонтальных линий, каждая со своим стилем концов (рис. 6.3).

Чтобы испытать этот код в действии, вставьте его в любую функцию JavaScript, а потом вызовите ее. Для немедленного исполнения сразу же после загрузки страницы вставьте его в функцию, которая обрабатывает событие `window.onload` (см. код шаблона страницы с холстом в *разд. "Базовые возможности холста" ранее в этой главе*).

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Устанавливаем толщину и цвет для всех линий.
context.lineWidth = 20;
context.strokeStyle = "rgb(205,40,40)";

// Рисуем первую линию с концами типа butt.
context.moveTo(10,50);
context.lineTo(400,50);
context.lineCap = "butt";
context.stroke();

// Рисуем вторую линию с концами типа round.
context.beginPath();
context.moveTo(10,120);
context.lineTo(400,120);
context.lineCap = "round";
context.stroke();
```

```
// Рисуем третью линию с концами типа square.
context.beginPath();
context.moveTo(10,190);
context.lineTo(400,190);
context.lineCap = "square";
context.stroke();
```

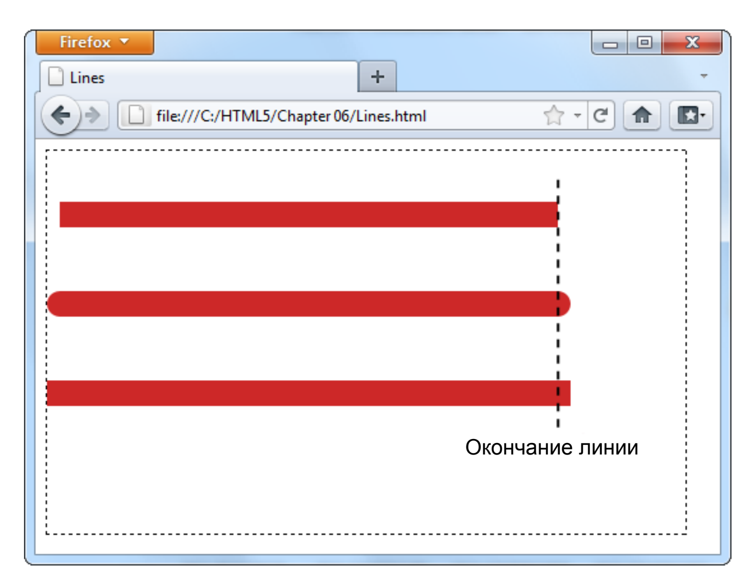


Рис. 6.3. Верхняя линия имеет стандартные прямоугольные концы типа `butt`, а на концы двух других добавлены наконечники (округлые и прямоугольные), которые удлиняют эти линии на половину их толщины

В этом примере вводится новая функция: метод `beginPath()` контекста рисования. Вызов метода `beginPath()` начинает новый, *отдельный путь рисунка*. Если не выполнять этот шаг, то при каждом новом вызове метода `stroke()` холст будет рисовать все снова. (Особенно это представляет проблему при изменении свойств контекста, когда рисование начинает выполняться по существующему контексту теми же фигурами, но с новым цветом, толщиной или окончаниями линий.)

ПРИМЕЧАНИЕ

Для того чтобы начать новый путь, надо вызвать метод `beginPath()`, а чтобы завершить путь, ничего особенного делать не нужно. Текущий путь автоматически считается завершенным, как только создается новый путь.

Пути и фигуры

В предыдущем примере, чтобы отделить линии друг от друга, для каждой из них создавался новый *путь*. Этот подход позволяет присвоить каждой линии свой цвет, а также толщину и тип окончания. Важность путей также состоит в том, что они позволяют заполнять цветом фигуры. Например, нарисуем красными линиями треугольник посредством следующего кода:

```
context.moveTo(250,50);
context.lineTo(50,250);
context.lineTo(450,250);
context.lineTo(250,50);

context.lineWidth = 10;
context.strokeStyle = "red";
context.stroke();
```

Теперь мы хотим закрасить внутреннюю область этого треугольника, но метод `stroke()` для этой задачи не подходит. Здесь нужно закрыть текущий путь с помощью метода `closePath()`, выбрать цвет заливки, установив значение свойства `fillStyle`, а потом вызвать метод `fill()`, чтобы собственно выполнить заливку:

```
context.closePath();
context.fillStyle = "blue";
context.fill();
```

В этом примере стоит сделать пару доводок. Первое: при закрытии пути нет надобности рисовать последний сегмент линии, т. к. вызов метода `closePath()` автоматически строит линию между последней нарисованной точкой и начальной точкой. Второе: лучше сначала выполнить заливку фигуры и только потом очертить ее контуры. В противном случае линии контура могут быть частично перекрыты заливкой.

Далее приведен полный код для рисования и заливки треугольника:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

context.moveTo(250,50);
context.lineTo(50,250);
context.lineTo(450,250);
context.closePath();

// Делаем заливку.
context.fillStyle = "blue";
context.fill();

// Рисуем контур.
context.lineWidth = 10;
context.strokeStyle = "red";
context.stroke();
```

Обратите внимание, что в этом примере метод `beginPath()` вызывать не нужно, т. к. путь создается автоматически. Метод `beginPath()` следует вызывать, только когда надо начать *новый* путь, например при изменении параметров линии или рисовании новой фигуры. Результаты выполнения кода показаны на рис. 6.4.

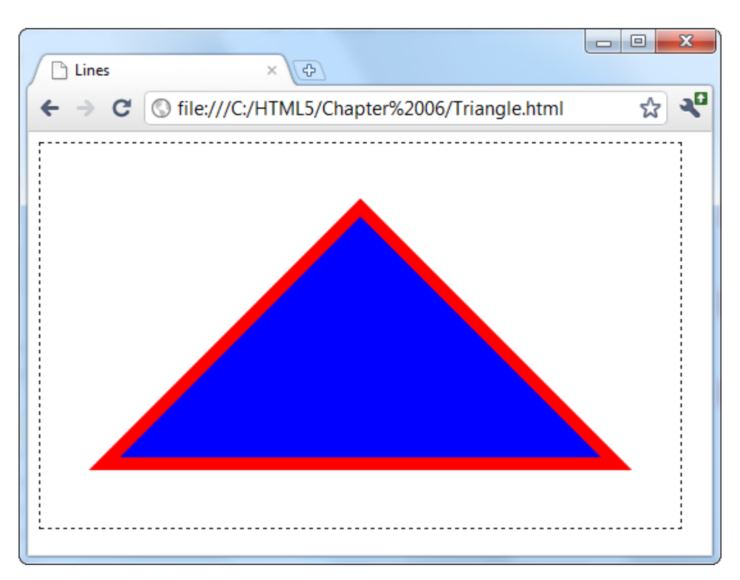


Рис. 6.4. Чтобы создать такую закрытую фигуру, устанавливаем начальную точку посредством метода `moveTo()`, указываем пути для каждой стороны с помощью метода `lineTo()`, а потом завершаем путь вызовом метода `closePath()`. Созданный таким образом путь можно заполнить цветом посредством метода `fill()` и очертить его контуры методом `stroke()`

Примечание

Вершины фигур, создаваемые соединяющимися линиями, можно оформить тремя разными способами, присваивая свойству контекста `lineJoin` соответствующие значения. Значение `round` округляет вершины, значение `mitre` соединяет линии в вершине "под ус", а значение `bevel` обрезает вершины прямой линией. По умолчанию свойству `lineJoin` присвоено значение `mitre`.

В большинстве случаев, чтобы создать сложную фигуру, ее очертания нужно создавать пошагово, по одному отрезку за раз. Но одна фигура является достаточно важной, и для нее выделен отдельный метод. Это прямоугольник. Заполнить прямоугольную область заливкой можно за один шаг методом `fillRect()`, которому в параметрах передаются координаты левого верхнего угла, ширина и высота прямоугольника.

Например, следующий код заполняет заливкой прямоугольную область размером 100 × 200 пикселов с левым верхним углом в точке (0, 10):

```
fillRect(0,10,100,200);
```

Цвет заливки для метода `fillRect()` устанавливается так же, как и для метода `fill()` свойством `fillStyle`.

В один присест, используя метод `strokeRect()`, также можно нарисовать и очертания прямоугольника:

```
strokeRect(0,10,100,200);
```

Толщина и цвет линий прямоугольника определяются текущими значениями свойств `lineWidth` и `strokeStyle`, точно так же, как для метода `stroke()`.

Кривые линии

Чтобы рисовать что-то более сложное, чем линии и прямоугольники, нужно изучить следующие четыре метода: `arc()`, `arcTo()`, `bezierCurveTo()` и `quadraticCurveTo()`. Все эти методы рисуют кривые линии, и хотя каждый делает это по-своему, все они требуют хотя бы небольших (а некоторые и больших) знаний математики.

Изо всех этих методов самый простой — метод `arc()`, который рисует дугу. Чтобы нарисовать дугу, нужно сначала представить себе в уме полный круг, а потом решить, какую часть его окружности вы хотите рисовать (рис. 6.5)[1].

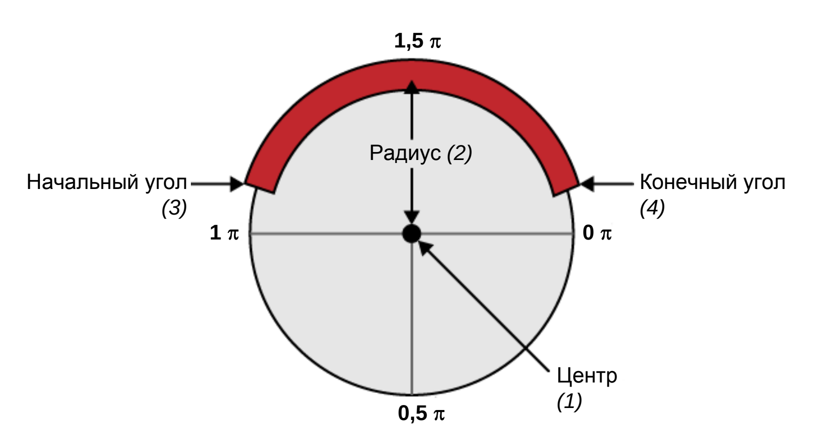


Рис. 6.5. Дуга выглядит достаточно простой фигурой, но чтобы полностью ее описать, требуется несколько единиц информации. Сначала нужно нарисовать воображаемый круг. Для этого надо знать координаты центра *(1)* и радиуса *(2)*, который определяет размер круга. Далее следует описать длину дуги на окружности, для чего требуется угол начала дуги *(3)* и угол ее окончания *(4)*. Значения углов должны быть в радианах, которые выражаются через число π. Угол всей окружности равен 2π, половины — 1π и т. д.

Собрав все необходимые данные, передаем их методу `arc()`:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Создаем переменные для хранения информации о дуге.
var centerX = 150;
var centerY = 300;
var radius = 100;
var startingAngle = 1.25 * Math.PI;
var endingAngle = 1.75 * Math.PI;

// Рисуем дугу на основе этой информации.
context.arc(centerX, centerY, radius, startingAngle, endingAngle);
context.stroke();
```

[1] Обратите внимание, что угол отсчитывается по часовой стрелке, а не против часовой, как принято в тригонометрии. — *Ред.*

Дугу можно закрыть, соединив ее концы прямой линией. Для этого нужно вызвать метод `closePath()` перед тем, как вызывать метод `stroke()`.

Кстати, окружность — это та же дуга, просто с углом 2π. Рисуется окружность следующим образом:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

var centerX = 150;
var centerY = 300;
var radius = 100;
var startingAngle = 0;
var endingAngle = 2 * Math.PI;

context.arc(centerX, centerY, radius, startingAngle, endingAngle);
context.stroke();
```

> ***Примечание***
>
> Метод `arc()` нельзя применять для рисования овала (вытянутого круга). Для этого нужно использовать либо более сложные методы для рисования кривых, которые рассматриваются далее в этой главе, либо применить трансформации, рассматриваемые в следующем, одноименном разделе.

Три других метода рисования кривых — `arcTo()`, `bezierCurveTo()` и `quadraticCurveTo()` — могут быть несколько посложнее для тех, кто не в ладах с геометрией. Они основаны на принципе *контрольных точек*, т. е. точек, которые сами не являются частью кривой, но управляют ее формой. Наиболее известным типом таких кривых являются кривые Безье, которые используются практически в каждой программе рисования. Причиной популярности этого метода является его способность создавать плавные кривые, независимо от их размера. На рис. 6.6 показано, как контрольные точки управляют формой кривой Безье.

Кривая Безье на рис. 6.6 создается следующим кодом:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Устанавливаем начало кривой.
context.moveTo(62, 242);

// Создаем переменные для контрольных точек и конечной точки
// и устанавливаем их значения.
var control1_x = 187;
var control1_y = 32;
var control2_x = 429;
var control2_y = 480;
var endPointX = 365;
var endPointY = 133;
```



Рис. 6.6. Кривая Безье имеет две контрольные точки. Сначала кривая Безье идет параллельно прямой, соединяющей начальную точку и первую контрольную точку (т. е. по направлению к первой контрольной точке), отходя от нее по мере удаления от начальной точки и приближения к конечной точке. По мере приближения к конечной точке, кривая приближается к линии, соединяющей конечную точку и вторую контрольную точку (т. е. входит в конечную точку). Степень изгиба кривой между двумя точками определяется расстоянием до контрольной точки — чем она дальше, тем сильнее ее притяжение или, вернее, отталкивание

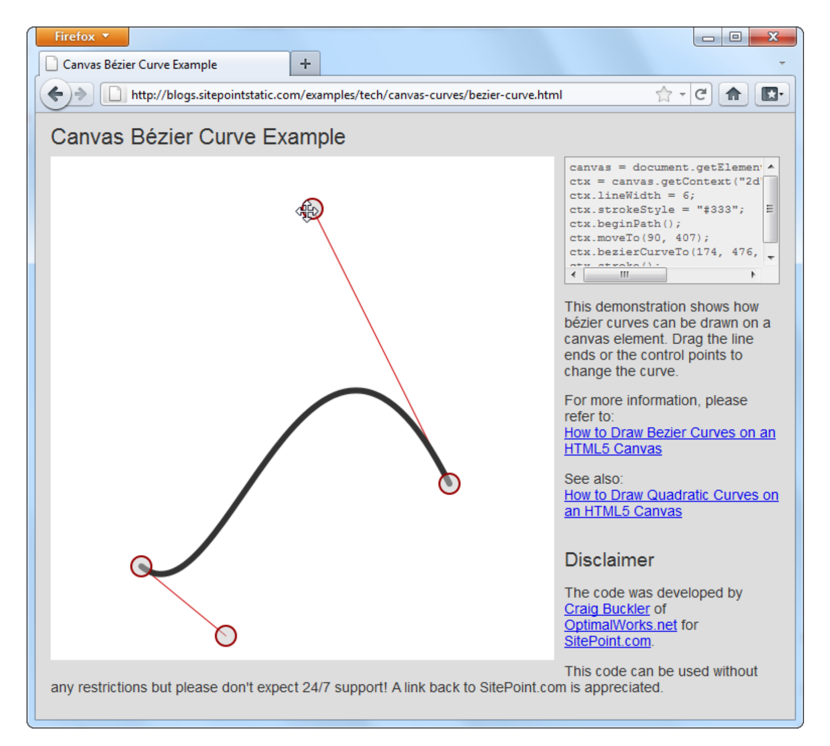


Рис. 6.7. На этой странице можно настраивать все аспекты кривой Безье, перетаскивая мышью ее точки. При перетаскивании точек страница генерирует их координаты, которые можно использовать для создания своей кривой. Подобную страницу для кривых второго порядка можно найти по адресу **http://tinyurl.com/html5quadratic**

```
// Рисуем кривую.
context.bezierCurveTo(control1_x, control1_y, control2_x,
                              control2_y, endPointX, endPointY);
context.stroke();
```

Контур сложной фигуры часто состоит из ряда дуг и кривых, соединяющихся друг с другом. По окончанию рисования всех составляющих можно вызвать метод `closePath()`, чтобы обвести или закрасить всю фигуру. Чтобы побольше узнать о кривых Безье, лучше поэкспериментировать с ними, а хорошую игровую площадку (рис. 6.7) можно найти вот по этому адресу: **http://tinyurl.com/html5bezier**.

Трансформации

Трансформация — это прием рисования, позволяющий перемещать систему координат холста. Допустим, нам нужно нарисовать один и тот же квадрат в трех местах холста. Это можно сделать, вызвав метод `fillRect()` три раза, каждый раз с другими координатами:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Рисуем квадрат размером 30x30 пикселов в трех разных местах.
context.rect(0, 0, 30, 30);
context.rect(50, 50, 30, 30);
context.rect(100, 100, 30, 30);

context.stroke();
```

ЧАСТО ЗАДАВАЕМЫЙ ВОПРОС

Рисование на холсте для тех, кто ненавидит математику

У меня от математики болит голова.
Можно ли рисовать фигуры без всех этих математических вычислений?

Если вы планируете создавать на холсте захватывающую графику, но не желаете углубленно изучать геометрию, вам, может быть, придется немного разочароваться. К счастью, существует несколько способов, которые могут помочь вам рисовать требуемые фигуры, не беспокоясь о математических принципах в их основе.

- **Использовать библиотеку рисования.** Зачем набивать себе шишки на лбу, если можно использовать готовую библиотеку для рисования кругов, треугольников, овалов и многоугольников за один прием? Идея простая — вызывается метод высшего уровня (скажем, `fillEllipse()`, которому передаются соответствующие координаты), и библиотека JavaScript преобразует все это в требуемую операцию на холсте. В качестве двух хороших примеров такой библиотеки можно назвать CanvasPlus (**http://code.google.com/p/canvasplus**) и Artisan JS (**http://artisanjs.com**). Но эти, и другие, библиотеки продолжают развиваться, поэтому еще рано говорить, какие из них выживут и окажутся пригодными для профессионального применения.
- **Рисовать растровые изображения.** Вместо того чтобы кропотливо рисовать каждую требуемую фигуру самому, можно скопировать в холст уже готовую графику. Например, изображение круга, сохраненное в файле формата PNG, можно вста-

вить в холст с помощью кода, рассматриваемого в *разд. "Вставка в холст изображений" главы 7*. Но этот подход не позволит использовать гибкость при манипулировании изображением, например растягивать его, перемещать или удалять его части и т. п.

- **Использовать профессиональный инструмент.** Для того чтобы манипулировать сложной графикой на холсте или сделать ее интерактивной, фиксированного растрового изображения будет недостаточно. Решением этой проблемы может быть средство преобразования, которое способно исследовать вашу графику и сгенерировать правильный код для применения с холстом. В этом отношении представляет интерес одно из таких средств — модуль Ai->Canvas для Adobe Illustrator (**http://visitmix.com/labs/ai2canvas**). Данный модуль преобразует созданную в Adobe Illustrator графику в страницу HTML с кодом JavaScript, который воспроизводит эту графику на холсте.

Тот же квадрат в трех разных местах можно нарисовать, используя те же его координаты, но сдвигая систему координат таким образом:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Рисуем квадрат с левым верхним углом в (0, 0).
   context.rect(0, 0, 30, 30);

// Сдвигаем систему координат вниз и вправо на 50 пикселов.
   context.translate(50, 50);
   context.rect(0, 0, 30, 30);

// Сдвигаем систему координат еще на 50 пикселов в каждом направлении.
// Преобразования аккумулируются, поэтому точка (0, 0)
// в действительности находится в точке (100, 100).
   context.translate(50, 50);
   context.rect(0, 0, 30, 30);
   context.stroke();
```

Оба варианта кода дают одинаковый результат: рисуют три квадрата в трех разных местах холста.

С первого взгляда может показаться, что трансформация делает *немного* сложную задачу рисования *еще более* сложной. Для обычных задач рисования может быть так оно и есть, но в некоторых особых ситуациях трансформация способна творить чудеса. Допустим, например, что у нас есть функция, рисующая последовательность сложных фигур, которые при сложении вместе образуют изображение птицы. Нам нужно анимировать это изображение, чтобы создать видимость полета птицы через экран. (Простой пример анимации на холсте рассматривается в *разд. "Анимация на холсте" главы 7*.)

Чтобы реализовать эту задачу, не прибегая к преобразованию системы координат, нужно было бы корректировать каждую координату птицы при каждом ее перемещении. Применение же трансформации позволяет использовать одни и те же координаты в коде рисования птицы и просто сдвигать систему координат для каждого шага анимации.

Существует несколько разных типов трансформаций. В предыдущем примере мы использовали *трансляцию*, или *перенос* (translation transform), чтобы переместить начальную точку системы координат, т. е. точку (0, 0) в левом верхнем углу холста. Кроме трансляции, существуют такие трансформации, как *масштабирование* (scale transform), которая позволяет рисовать в увеличенном или уменьшенном масштабе; *вращение* (rotate transform), позволяющая вращать систему координат; и *матричная трансформация* (matrix transform), позволяющая растягивать и сгибать систему координат практически как угодно при условии, что вы понимаете сложную матричную математику в основе требуемых визуальных эффектов.

Трансформации обладают эффектом аккумулирования. В следующем примере начало координат (т. е. точка (0, 0)) перемещается в точку (100, 100) посредством трансляции, после чего система координат поворачивается несколько раз на определенный угол вокруг этой точки. При каждом вращении рисуется новый квадрат, создавая фигуру, показанную на рис. 6.8.

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Перемещаем точку (0, 0). Это важный шаг,
// т. к. вращение выполняется вокруг этой точки.
context.translate(100, 100);

// Рисуем 10 квадратов.
var copies = 10;
for (var i=1; i<copies; i++) {
  // Прежде чем рисовать следующий квадрат, вращаем систему координат.
  // Угол полного оборота составляет 2*Math.PI. Этот код вращает систему
  // координат для каждого нового квадрата лишь на долю этого угла,
  // выполняя полный оборот при рисовании последнего квадрата.
  context.rotate(2 * Math.PI * 1/(copies-1));

  // Рисуем квадрат.
  context.rect(0, 0, 60, 60);
}
context.stroke();
```

Совет

С помощью метода контекста `save()` можно сохранить текущее состояние системы координат и восстановить его позже посредством метода `restore()`. Сохраняйте состояние контекста перед тем, как применять трансформацию, чтобы в случае ошибки можно было восстановить систему координат. А при рисовании сложных фигур, требующих длинной последовательности шагов, разумно сохранять состояние как много чаще. Этот список сохраненных состояний действует подобно истории посещенных веб-страниц. При каждом вызове метода `restore()` система возвращается в непосредственно предшествующее состояние.

Более углубленное рассмотрение предмета трансформаций выходит за рамки этой книги. Если же вы желаете исследовать этот предмет более подробно, можете найти полезную документацию и примеры на сайте **http://tinyurl.com/b742o4**.

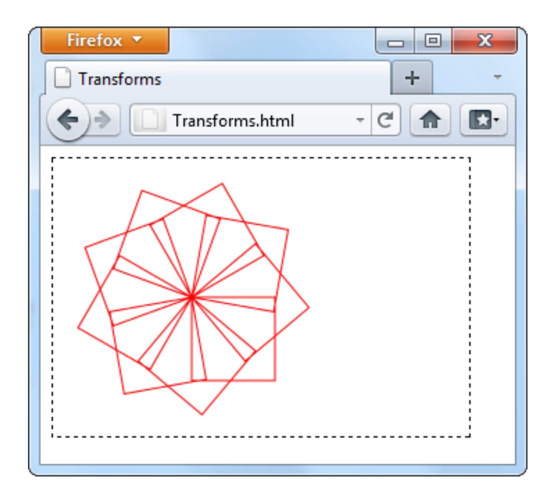


Рис. 6.8. Результат рисования последовательности квадратов с одними и теми же координатами, но с системой координат, вращающейся вокруг одной точки

Прозрачность

До сих пор мы имели дело с чистыми цветами. Но холст также позволяет применять частичную прозрачность, чтобы накладывать одну фигуру поверх другой. Существуют два способа использования прозрачности. Первый позволяет установить цвет, присвоив значения свойству `fillStyle` или `strokeStyle` и используя функцию `rgba()`, а не более распространенную функцию `rgb()`. Функция `rgba()` принимает три параметра — значения красной, зеленой и синей цветовых составляющих (от 0 до 255) и дополнительное значение альфа (от 0 до 1), которое указывает прозрачность цвета. Значение альфа, равное 1, соответствует полной непрозрачности, а значение, равное 0 — полной прозрачности. Установив значение альфа между этими двумя пределами (например, 0,5), мы получим частично прозрачный цвет, позволяющий просматривать любое находящееся под ним содержимое.

Примечание

Местонахождение содержимого — снизу или сверху — определяется порядком операций рисования. Например, если в одном и том же месте сначала нарисовать круг, а потом квадрат, квадрат будет наложен на круг.

В следующем листинге приведен код для рисования круга и треугольника. Обе фигуры закрашиваются одним и тем же цветом, только для треугольника значение альфа устанавливается равным 0,5, делая его на 50% прозрачным.

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Устанавливаем цвета заливки и очертаний.
  context.fillStyle = "rgb(100,150,185)";
  context.lineWidth = 10;
  context.strokeStyle = "red";
```

```
// Рисуем круг.
  context.arc(110, 120, 100, 0, 2*Math.PI);
  context.fill();
  context.stroke();

// Не забудьте вызвать метод beginPath() перед тем,
// как рисовать новую фигуру.
// В противном случае фигуры могут быть совмещены.
  context.beginPath();

// Заполняем треугольник прозрачным цветом.
  context.fillStyle = "rgba(100,150,185,0.5)";

// Теперь рисуем треугольник.
   context.moveTo(215,50);
   context.lineTo(15,250);
   context.lineTo(315,250);
   context.closePath();
   context.fill();
   context.stroke();
```

Результаты выполнения кода показаны на рис. 6.9.

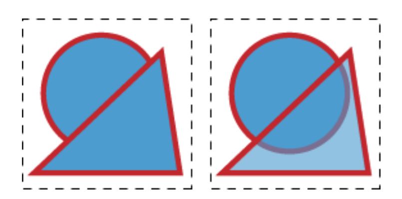


Рис. 6.9. *Слева:* две фигуры, закрашенные сплошным цветом, треугольник наложен поверх круга. *Справа:* круг закрашен сплошным цветом с наложенным полупрозрачным треугольником. Полупрозрачные фигуры выглядят светлее (т. к. они пропускают сквозь себя белый фон) и позволяют видеть расположенное под ними содержимое. Обратите внимание, что в этом примере полупрозрачный треугольник имеет полностью непрозрачные границы

Другой способ использования прозрачности — установить значение свойства контекста `globalAlpha`, как показано в следующем коде:

```
context.globalAlpha = 0.5;
// Теперь этому цвету автоматически присваивается значение альфа,
// равное 0,5
context.fillStyle = "rgb(100,150,185)";
```

После этого вся графика будет иметь одинаковое значение альфа и одинаковый уровень прозрачности, пока значению `globalAlpha` не будет присвоено новое значение. Это включает как цвета очертаний (свойство `strokeStyle`), так и цвета заливки (свойство `fillStyle`).

Какой из этих подходов лучше? Если требуется один прозрачный цвет, используйте функцию `rgba()`. А если требуется нарисовать несколько фигур разного цвета, но

с одинаковым уровнем прозрачности, используйте свойство `globalAlpha`. Свойство `globalAlpha` также полезно в том случае, когда нужно рисовать полупрозрачные изображения на холсте *(см. разд. "Вставка в холст изображений" главы 7)*.

ПРАКТИЧЕСКИЕ ЗАНЯТИЯ ДЛЯ ОПЫТНЫХ ПОЛЬЗОВАТЕЛЕЙ

Составные операции

До сих пор мы полагали, что при наложении одной фигуры поверх другой верхняя фигура рисуется поверх нижней, скрывая ее. В большинстве случаев так оно и есть, но холст также поддерживает более сложные *составные операции*.

Составная операция — это правило, указывающее холсту, каким образом отображать два перекрывающихся изображения. Составной операцией по умолчанию является операция `source-over` (источник сверху), при которой вторая фигура рисуется поверх первой. Но возможны и другие типы составных операций. Например, операция `xor` не показывает ничего в области перекрытия изображений. На рис. 6.10 показаны результаты применения составных операций рисования.

Тип используемой холстом составной операции меняется посредством присвоения соответствующего значения свойству контекста `globalCompositeOperation`:

```
context.globalCompositeOperation = "xor";
```

При умелом применении составные операции могут значительно сократить определенные процедуры рисования. К сожалению, среди браузеров нет согласия, как отображать эти операции, в результате чего некоторые составные операции отображаются по-разному на различных браузерах. Дополнительную информацию об этой проблеме и сравнение отличий в отображении составных операций см. в блоге по адресу **http://tinyurl.com/68b2nmz**.

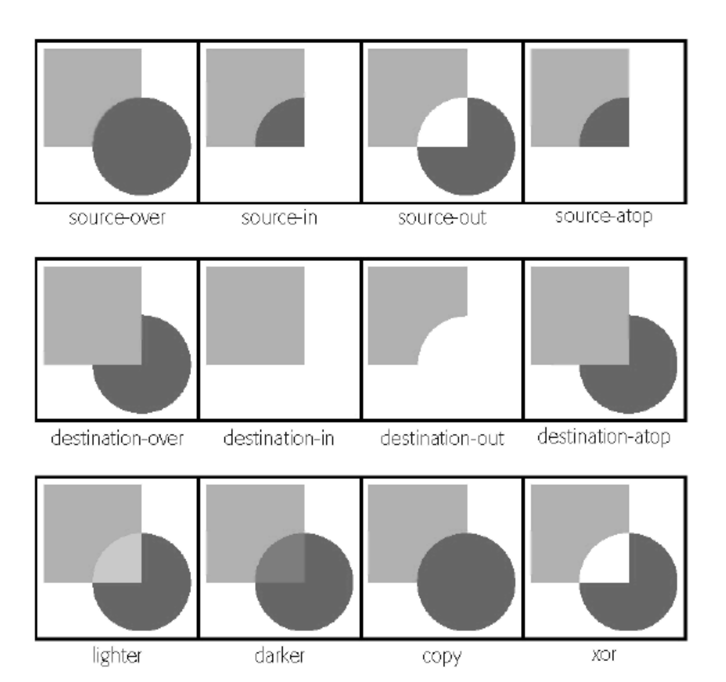


Рис. 6.10. Разные типы составных операций рисования, как они отображаются в браузере Firefox. Браузеры Internet Explorer 9 и Opera обрабатывают операцию `copy` иначе, а браузеры Chrome и Safari обрабатывают по-разному операции `source-in`, `source-out`, `destination-in` и `destination-atop`

Создание простой программы рисования

Кроме рассмотренных, холст имеет много других возможностей. Но на данном этапе вы обладаете достаточным объемом знаний, чтоб создать простою программу рисования на основе холста (рис. 6.11).

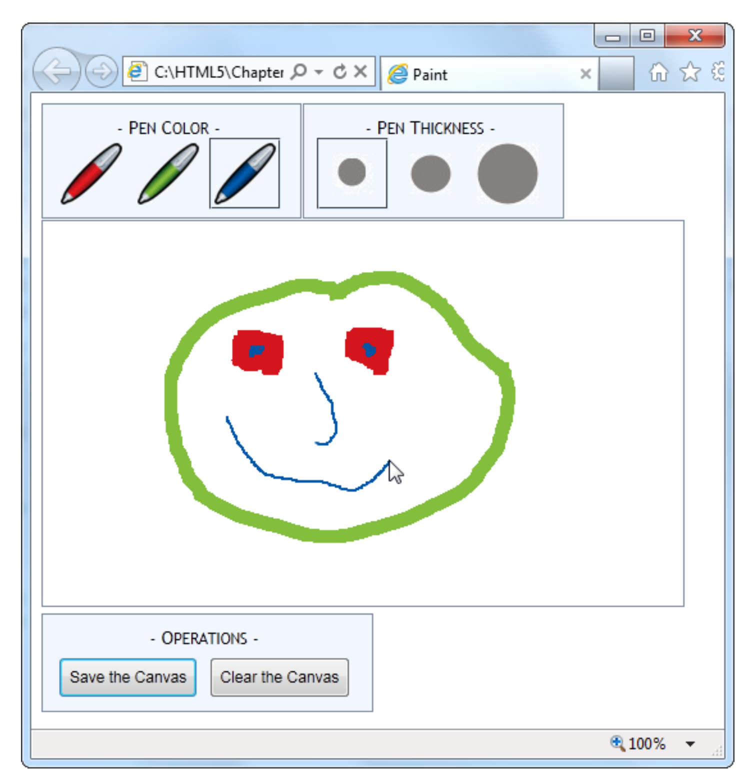


Рис. 6.11. Для работы с этой программой рисования выберите цвет и толщину линии, а потом рисуйте с помощью мыши

Код JavaScript для этой программы более объемный, чем в предыдущих примерах, но все равно на удивление простой. Мы рассмотрим эту программу по частям в следующих разделах.

Совет

Если вы хотите знать правила таблицы стилей, с помощью которых создаются панели инструментов вверху и внизу поля для рисования, или желаете исследовать весь код примера, или просто хотите сразу же испытать программу на практике, все это можно сделать с помощью файла Paint.html на сайте этой книги (**www.prosetech.com/html5**).

Подготовка к рисованию

При загрузке страницы код получает объект холста `canvas` и подключает к нему функции для обработки нескольких событий JavaScript для разных движений мыши: `onMouseDown`, `onMouseUp`, `onMouseOut` и `onMouseMove`. (Как мы увидим далее, эти

события управляют процессом рисования.) В то же самое время страница сохраняет холст в глобальной переменной `canvas`, а контекст рисования — в другой глобальной переменной, `context`. Таким образом, эти объекты будут доступны для остального кода:

```
var canvas;
var context;

window.onload = function() {
  // Получаем объект холста и контекст рисования.
  canvas = document.getElementById("drawingCanvas");
  context = canvas.getContext("2d");

  // Подключаем требуемые для рисования события.
  canvas.onmousedown = startDrawing;
  canvas.onmouseup = stopDrawing;
  canvas.onmouseout = stopDrawing;
  canvas.onmousemove = draw;
};
```

Чтобы начать рисовать, пользователь выбирает цвет и толщину линии, щелкнув по требуемым значкам в панели инструментов вверху окна рисования. Эти панели инструментов создаются с помощью простых элементов `<div>`, отформатированных под светло-голубой фон и содержащих по несколько элементов `<img>`, активизируемых щелчком мыши. Например, код, создающий панели инструментов для выбора цвета, выглядит так:

```
<div class="Toolbar">
  — Pen Color -<br>
  <img id="redPen" src="pen_red.gif" alt="Red Pen"
       onclick="changeColor('rgb(212,21,29)', this)">
  <img id="greenPen" src="pen_green.gif" alt="Green Pen"
       onclick="changeColor('rgb(131,190,61)', this)">
  <img id="bluePen" src="pen_blue.gif" alt="Blue Pen"
       onclick="changeColor('rgb(0,86,166)', this)">
</div>
```

Важной частью этой разметки является атрибут `onclick` элемента `<img>`. При щелчке пользователя по значку ручки определенного цвета элемент `<img>` вызывает функцию `changeColor()`. Эта функция принимает два параметра — новый цвет, который совпадает с цветом выбранного значка, и ссылки на элемент `<img>`, по которому щелкнули. Код функции выглядит так:

```
// Отслеживает элемент <img>, по которому ранее щелкнули, для цвета.
var previousColorElement;

function changeColor(color, imgElement) {
  // Меняем текущий цвет рисования.
  context.strokeStyle = color;
```

```
  // Меняем стиль элемента <img>, по которому щелкнули, для цвета.
  imgElement.className = "Selected";

  // Возвращаем ранее элемент <img>, по которому ранее щелкнули,
  // в нормальное состояние.
  if (previousColorElement != null) previousColorElement.className = "";
      previousColorElement = imgElement;
}
```

Код функции `changeColor()` выполняет две основные задачи. Во-первых, он устанавливает свойство контекста `strokeStyle` в значение соответствующего нового цвета. Для этого требуется всего лишь одна строка кода. Во-вторых, код изменяет оформление элемента `<img>`, по которому щелкнули, заключая его в рамку, чтобы было видно, какой цвет является текущим. Выполнение этой операции требует больше работы, т. к. необходимо отслеживать цветовой элемент `<img>`, по которому ранее щелкнули, чтобы удалить его рамку.

Функция `changeThickness()` выполняет почти идентичную работу, только она изменяет значение свойства контекста `lineWidth` в соответствии с выбранной толщиной линии:

```
// Отслеживаем элемент <img>, по которому ранее щелкнули,
// для толщины линии.
var previousThicknessElement;

function changeThickness(thickness, imgElement) {
  // Изменяем текущую толщину линии.
  context.lineWidth = thickness;

  // Меняем стиль элемента <img>, по которому щелкнули, для толщины цвета.
  imgElement.className = "Selected";

  // Возвращаем элемент <img>, по которому ранее щелкнули,
  // для толщины линии в нормальное состояние.
  if (previousThicknessElement != null) {
      previousThicknessElement.className = "";
  }
  previousThicknessElement = imgElement;
}
```

Рассмотренный выше код подготавливает холст для рисования, но рисовать еще рано. Для этого нужно добавить код, который собственно выполняет рисование, что и делается в следующем разделе.

Рисование на холсте

Процесс рисования начинается, когда пользователь щелкает мышью по холсту. Для отслеживания, когда осуществляется рисование, в программе используется гло-

бальная переменная isDrawing, информирующая остальной код программы, можно ли работать с контекстом рисования.

Как мы видели ранее, событие onMouseDown связано с функцией startDrawing(). Эта функция устанавливает переменную isDrawing, создает новый путь, а потом устанавливает начальную позицию рисования, подготовив таким образом холст для рисования:

```
var isDrawing = false;

function startDrawing(e) {
  // Начинаем рисовать.
  isDrawing = true;

  // Создаем новый путь (с текущим цветом и толщиной линии).
  context.beginPath();

  // Нажатием левой кнопки мыши помещаем "кисть" на холст.
  context.moveTo(e.pageX-canvas.offset Left, e.pageY-canvas.offsetTop);
}
```

Чтобы наша программа работала корректно, рисование должно начинаться в текущей позиции, т. е. там, где находится указатель мыши, когда пользователь нажимает левую кнопку. Но задача получения правильных координат этой точки несколько сложновата.

Событие onMouseDown предоставляет координаты курсора мыши (через свойства pageX и pageY, показанные в этом примере), но это координаты относительно всей страницы. Чтобы вычислить соответствующие координаты холста, необходимо от координат левого верхнего угла окна браузера отнять расстояние до левого верхнего угла холста.

Собственно рисование происходит, когда пользователь двигает мышь, удерживая нажатой левую кнопку. При каждом перемещении мыши, даже на один пиксел, активируется событие onMouseMove и исполняется код функции draw(). Если переменная isDrawing установлена, код вычисляет текущие координаты холста (т. е. координаты точки, в которой в данный момент находится указатель мыши), а потом вызывает метод lineTo(), который добавляет путь соответствующего отрезка линии, после чего вызывается метод stroke(), который прорисовывает эту линию:

```
function draw(e) {
  if (isDrawing == true) {
    // Определяем текущие координаты указателя мыши.
    var x = e.pageX — canvas.offsetLeft;
    var y = e.pageY — canvas.offsetTop;
    // Рисуем линию до новой координаты.
    context.lineTo(x, y);
    context.stroke();
  }
}
```

Если пользователь продолжает перемещать мышь, снова вызывается функция `draw()`, опять добавляя отрезок к уже нарисованной линии. Этот отрезок настолько короткий — длиной всего лишь в один или два пиксела, что он даже не выглядит, как прямая линия в начале рисования.

Наконец, когда пользователь отпускает кнопку мыши или выводит курсор за пределы холста, срабатывает событие `onMouseUp` или `onMouseOut` соответственно. Оба эти события активируют функцию `StopDrawing()`, которая указывает приложению прекратить рисование:

```
function stopDrawing() {
  isDrawing = false;
}
```

На данном этапе мы рассмотрели все аспекты рисования. Теперь перейдем к обсуждению функций очистки холста или сохранения созданного рисунка. Для этих целей предназначены две кнопки на панели инструментов **Operations** внизу холста — **Clear the Canvas** и **Save the Canvas** соответственно. Нажатие кнопки **Clear the Canvas** вызывает функцию `clearCanvas()`, которая полностью очищает поверхность холста с помощью метода контекста `clearRect()`:

```
function clearCanvas() {
  context.clearRect(0, 0, canvas.width, canvas.height);
}
```

Операция сохранения содержимого холста более сложна, и мы посвятим ей отдельный раздел.

Сохранение содержимого холста

Задача сохранения содержимого холста требует рассмотрения множества опций. Прежде всего, нужно решить, каким образом получить данные рисунка. Для решения этой задача холст предоставляет три возможных подхода.

- **Использовать URL данных.** При этом подходе содержимое холста преобразуется в файл изображения, которое потом переводится в последовательность символов, оформленных в виде URL. Это позволяет получить аккуратный и компактный способ для перемещения данных изображения (например, его можно передать элементу `<img>` и отослать на веб-сервер). В нашей программе рисования используется этот подход.
- **Использовать метод `getImageData()`.** Этот подход позволяет получить "сырые" пиксельные данные, которыми можно потом манипулировать как угодно. Этот подход рассматривается в *разд. "Проверка на столкновение с использованием цвета пикселов" главы 7*.
- **Сохранять список "шагов".** Например, можно организовать массив, содержащий список всех линий, нарисованных на холсте. Эти данные можно потом сохранить и использовать для воспроизведения изображения. Данный подход требует меньше места для хранения изображения, а также предоставляет боль-

шую гибкость для последующей работы с ним. К сожалению, для этого нужно отслеживать все выполняемые шаги, используя метод, который рассматривается в *разд. "Отслеживание нарисованного содержимого" главы 7*.

Если все это кажется слегка пугающим, это еще не все. Определившись с типом содержимого для сохранения, нужно решить, *где* это содержимое сохранить. Возможны, среди прочих, следующие опции.

- **В файле изображения.** Например, содержимое холста сохраняется в виде файла формата PNG или JPEG на локальном жестком диске художника. Это подход, который применяется в нашей программе рисования, и мы его рассмотрим далее в этом разделе.
- **В локальной системе хранения.** Работа этого подхода рассматривается в *главе 9*.
- **На веб-сервере.** После передачи данных веб-серверу последний может сохранить их в файле или базе данных и предоставить при следующем посещении страницы пользователем.

Для сохранения содержимого холста в нашей программе рисования применяется возможность, которая называется *URL данных* (data URL). Чтобы получить URL для текущих данных, мы просто используем метод холста `toDataURL()`:

```
var url = canvas.toDataURL();
```

Если вызвать метод `toDataURL()`, не передав ему никаких параметров, то получим изображение в формате PNG. Альтернативно, методу можно указать требуемый формат изображения:

```
var url = canvas.toDataURL("image/jpeg");
```

Но если браузер не может предоставить требуемый формат, он опять выдаст изображение в формате PNG, преобразованное в длинную строку.

Что же собой представляет URL данных? Технически это просто длинная строка символов, закодированных алгоритмом Base64, которая начинается с текста: `data:image/png;base64`. Это выглядит как бессмыслица, по крайней мере, для людей, т. к. эти данные предназначены для понимания компьютерными программами, например браузерами. URL данных для содержимого холста на рис. 6.11 выглядит таким образом:


```
data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAfQAAAEsCAYAAAA1u0HIAAAAAXNSR0IAr
s4c6QAAAARnQU1BAACxjwv8YQUAACqRSURBVHhe7Z1bkB1Hecdn5uxFFzA2FWOnsEEGiiewnZgKsrWL
rZXMRU9JgZQKhoSHVK...gAAEIQAACEIBAiAT+HxAYpeqDfKieAAAAAElFTkSuQmCC
```


Следует заметить, что с целью экономии книжного пространства, в этом примере URL данных пропущена огромная часть данных в средине, представленная многоточием. В полном виде URL данных этого примера занял бы пять страниц книги.

Примечание

Алгоритм кодирования Base64 преобразует данные изображения в длинную последовательность букв, цифр и некоторых специальных символов. Знаки пунктуации и необычные расширенные символы не используются, в результате чего эти данные можно без проблем вставлять в веб-страницы (например, чтобы установить значение скрытого поля ввода или атрибут `src` элемента `<img>`).

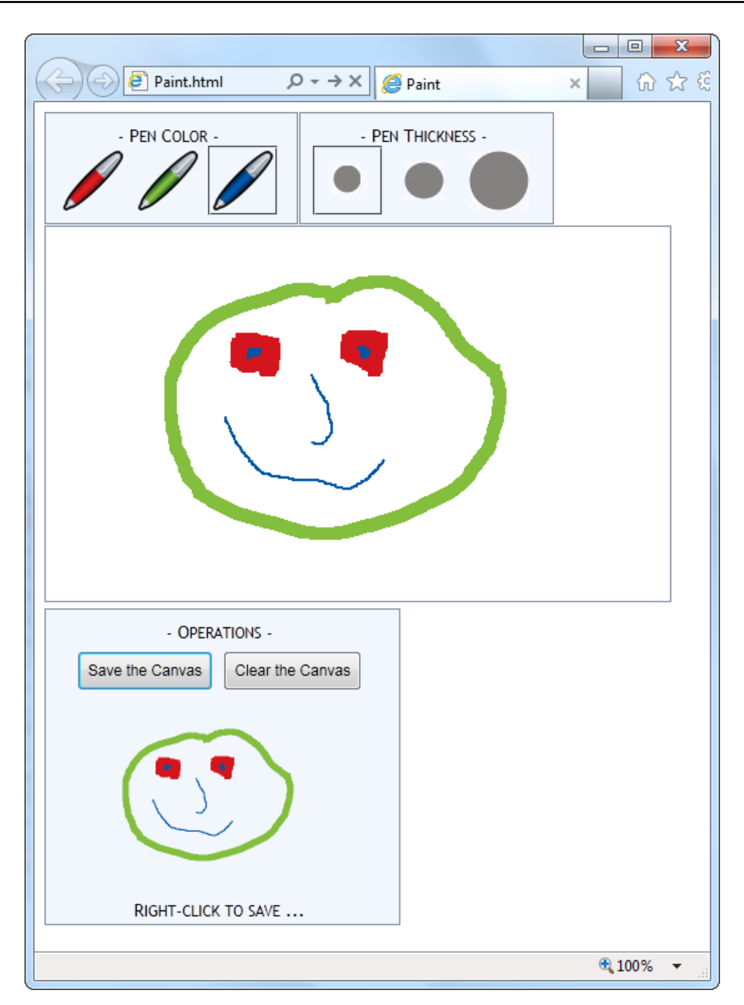


Рис. 6.12. Данные изображения из холста передаются в элемент <img> с помощью URL. Поле элемента <img> сделано меньшего размера, чтобы изображение в нем отличалось от основного изображения на холсте. Чтобы сохранить это изображение в файл формата PNG, просто щелкните на нем правой кнопкой мыши и в контекстном меню выберите команду **Сохранить картинку как**, точно так же, как и для сохранения изображения из обычной веб-страницы

Таким образом, задача преобразования данных изображения в URL данных не представляет никаких трудностей. Но что можно делать дальше с этим URL данных? Можно отправить его на веб-сервер для длительного хранения. Пример выполнения этой операции с помощью кода PHP представлен на веб-странице **http://tinyurl.com/5uud9ob**.

Но возможности для сохранения данных на стороне клиента несколько ограничены. Некоторые браузеры позволяют перейти непосредственно к URL данных. Это означает, что для перехода к изображению можно использовать следующий код:

```
window.location = canvas.toDataURL();
```

Более надежным методом будет передать URL данных в элемент <img>, что и делает наша программа рисования:

```
function saveCanvas() {
  // Находим элемент <img>.
  var imageCopy = document.getElementById("savedImageCopy");

  // Отображаем данные холста в элементе <img>.
  imageCopy.src = canvas.toDataURL();

  // Показываем элемент <div>, содержащий элемент <img>,
  // делая изображение видимым.
  var imageContainer = document.getElementById("savedCopyContainer");
  imageContainer.style.display = "block";
}
```

Этот код не совсем сохраняет данные изображения, т. к. изображение еще не сохранено в файле. Но для этого требуется всего лишь один шаг — просто щелкнуть правой кнопкой мыши по уменьшенному изображению в панели под холстом и в открывшемся контекстном меню выбрать команду **Сохранить картинку как**. Это не так удобно, как закачка файла или диалоговое окно **Сохранить**, но единственный способ, который надежно работает во всех браузерах.

Примечание

Браузер Firefox имеет встроенную возможность сохранения содержимого холста. Для этого щелкнуть правой кнопкой нужно не на копии изображения в панели под холстом, а на самом изображении в холсте, и в открывшемся контекстном меню выбрать команду **Сохранить как**.

Примечание

Возможность URL данных является одной из возможностей холста, которая может не работать при открытии страницы с локального жесткого диска. Во избежание этой проблемы, для тестирования своей работы следует выгрузить ее на веб-сервер.

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ

Программы рисования на холсте, доступные в Интернете

Программа рисования — это первое, что приходит на ум многим людям, когда они начинают применять на практике свои знания программирования холста. Поэтому неудивительно, что с помощью Google в Интернете можно найти большое число таких программ, и некоторые из них на удивление продвинутые[1]. В качестве примера, можно назвать программы рисования на следующих веб-сайтах:

- **www.jswidget.com/index-ipaint.html.** Более продвинутая программа рисования, с панелью инструментов наподобие ленты Microsoft Word 2010. По завершению этого эксперимента его автор планирует приступить к разработке клона Adobe Photoshop в браузере;
- **http://mugtug.com/sketchpad.** Программа для рисования и черчения, изобилующая разнообразными эффектами и продвинутыми возможностями, такими как ма-

[1] Программы рисования можно найти и в русскоязычном секторе Интернета. Наберите в поисковой системе фразы вроде "графический редактор онлайн на HTML5" или "графический редактор интерактивный в HTML5", и вы сразу увидите ссылки на соответствующие веб-страницы. — *Ред.*

нипулирование изображениями, выделение прямоугольником и рисование спирографов[1]. В настоящее время эта программа не работает ни на одной из версий Internet Explorer.

Совместимость холста с браузерами

На данный момент мы многое узнали о холсте и многому научились. Настало время сделать шаг назад и ответить на вопрос, который дамокловым мечом висит над каждой новой возможностью HTML5: насколько безопасно использование этой возможности?

К счастью, холст является одной из наиболее поддерживаемых возможностей HTML5. Она поддерживается всеми последними версиями основных браузеров (табл. 6.1). Конечно же, чем выше версия браузера, тем выше уровень поддержки, т. к. в более новых версиях браузеров увеличивается скорость прорисовки и исправляются ошибки.

Таблица 6.1. *Поддержка холста основными браузерами*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
Минимальная версия	9	3.5	3	4	10	1	1

Вряд ли вам придется столкнуться с более старой версией любого из этих браузеров, за исключением Internet Explorer. Но это одно исключение ставит использующих холст разработчиков перед вопросом: как можно использовать холст в веб-страницах так, чтобы не лишать возможности просматривать подобные страницы пользователей старых, но все еще популярных версий этого браузера, таких как IE 8 и IE 7?

Как и в случае со многими другими возможностями HTML5, к решению этой проблемы совместимости есть два подхода. Первый — определять отсутствие поддержки холста браузером и пытаться использовать другое решение. А второй подход — закрыть этот пробел другим инструментом, способным эмулировать холст HTML5, чтобы страницы работали без изменений на старых браузерах. В случае холста второй подход неожиданно оказывается лучшим, как мы увидим в следующем разделе.

Холст на заполнителе

Чтобы позволить старым версиям Internet Explorer работать с холстом, существует несколько надежных резервных решений. Одним из таких решений является библиотека ExplorerCanvas (также называется excanvas), разработанная инженером из

[1] См. **http://ru.wikipedia.org/wiki/Спирограф_(игрушка)**. — *Прим. пер.*

Google и гением JavaScript Эриком Арвидссоном (Erik Arvidsson). Эта библиотека эмулирует холст в Internet Explorer 7 и 8, используя только JavaScript и устаревшую в настоящее время технологию VML (Vector Markup Language, язык векторной разметки).

ПРИМЕЧАНИЕ

VML представляет собой спецификацию для создания штриховых рисунков и другой графики в документе HTML с использованием разметки. Эта технология была заменена похожим, но с лучшим уровнем поддержки, стандартом SVG (Scalable Vector Graphics, масштабируемая векторная графика), который браузеры только начинают поддерживать. В настоящее время технология VML все еще продолжает использоваться в нескольких продуктах корпорации Microsoft, таких как Microsoft Office и Internet Explorer. Это позволяет использовать ее в качестве достаточно приемлемого заменителя холста, хотя и с некоторыми ограничениями.

Последнюю версию библиотеки ExplorerCanvas можно загрузить по адресу **http://code.google.com/p/explorercanvas**. Скопируйте файл excanvas.js в папку веб-страницы и добавьте в веб-страницу следующую ссылку на этот файл:

```
<head>
  <title>...</title>
  <!--[if lt IE 9]>
    <script src="excanvas.js"></script>
  <![endif]-->
  ...
<head>
```

Обратите внимание, что эта ссылка является условной. Версии Internet Explorer более ранние, чем IE 9, будут использовать ее, но IE 9 и другие браузеры не будут вообще обращать на нее внимания.

Теперь элемент `<canvas>` можно использовать без особых проблем. Например, эта библиотека — все что требуется, чтобы наша программа рисования *(см. разд. "Создание простой программы рисования" ранее в этой главе)* работала в старых версиях Internet Explorer.

ПРИМЕЧАНИЕ

Для работы на холсте с текстом *(см. разд. "Вставка в холст текста" главы 7)* нужна еще одна библиотека JavaScript — Canvas-text, которая работает совместно с библиотекой ExplorerCanvas. Загрузить эту библиотеку можно по адресу **http://code.google.com/p/canvas-text**.

К сожалению, библиотека ExplorerCanvas — не совсем идеальное решение. Если вы попытаетесь использовать более сложные приемы рисования, существует вероятность, что некоторые детали не будут выглядеть так, как вы ожидаете. Основные возможности, не поддерживаемые в ExplorerCanvas (на момент написания этой книги, в любом случае), включают радиальные градиенты, тени, обрезку изображений, обработку "сырых" пикселов, а также URL данных.

Для по-настоящему амбициозных задумок, например сложных анимаций или игр с горизонтальной прокруткой экрана (side-scrolling), ExplorerCanvas может оказаться

недостаточно быстрой. В такой ситуации следует рассмотреть возможность применения другого заполнителя, использующего высокопроизводительный модуль расширения, такой как Silverlight или Flash. Исследовать доступные опции можно на странице заполнителей по адресу **http://tinyurl.com/polyfills**. Или же сразу используйте лучший из них — бесплатную библиотеку FlashCanvas, которую можно загрузить по адресу **http://code.google.com/p/flashcanvas**. Подобно библиотеке ExplorerCanvas, эту библиотеку можно подключить к веб-странице с помощью одной строки кода JavaScript. Но в отличие от ExplorerCanvas она использует модуль Flash, без малейшего следа VML.

Библиотека FlashCanvas также имеет и профессиональную коммерческую версию — FlashCanvas Pro (**http://flashcanvas.net/purchase**). Эта версия предоставляет еще лучший уровень поддержки холста, который, согласно результатам тестирования браузерной совместимости, примерно такой же, как и уровень поддержки холста большинством основных браузеров. Сравнить уровень браузерной поддержки, предоставляемой этими тремя разными библиотеками — ExplorerCanvas, FlashCanvas и FlashCanvas Pro — можно по адресу **http://flashcanvas.net/docs/canvas-api**.

Примечание

Совместив на странице возможности холста и библиотеки наподобие FlahsCanvas, вы получите действительно замечательную поддержку практически всех браузеров, известных человечеству в настоящее время. Таким образом, вы обеспечите поддержку не только для немного устарелых версий Internet Explorer, посредством использования Flash, но также и для мобильных устройств, не использующих Flash, таких как iPad и iPhone, с помощью HTML5.

Резервное решение для холста и определение возможностей

Самый популярный способ расширить доступность страниц, использующих холст — это применить библиотеки ExplorerCanvas и FlashCanvas. Но они не являются единственными возможными опциями.

Как и элементы `<audio>` и `<video>`, элемент `<canvas>` поддерживает резервное содержимое. Например, следующая разметка позволяет либо использовать холст (если он поддерживается браузером), либо выводить изображение (если холст не поддерживается):

```
<canvas id="logoCreator" width="500" height="300">
  <p>Ваш компьютер не поддерживает возможности холста,
     поэтому вы не можете использовать наш генератор
     динамических логотипов.</p>
  <img src="logo.png" alt="Стандартный логотип компании">
</canvas>
```

Но этот метод редко бывает полезным. В большинстве случаев холст используется для рисования динамической графики или для создания какого-либо интерактивного приложения, для которых статическое изображение не будет удовлетворитель-

ным заменителем. Одно из альтернативных решений — вставить в элемент `<canvas>` приложение Flash. Этот подход особенно хорош в тех случаях, когда уже имеется Flash-версия страницы, но на перспективу к ней нужно добавить возможности холста. Он предлагает работоспособное Flash-решение для старых версий Internet Explorer и в то же время позволяет остальным браузерам использовать холст без модулей расширения.

При использовании Modernizr *(см. разд. "Определение возможностей с помощью Modernizr" главы 1)* можно проверить поддержку холста браузером с помощью JavaScript. Для проверки на поддержку рисования нужно исследовать значение свойства `Modernizr.canvas`, а работы с текстом — свойства `Modernizr.canvastext`. В случае если браузер не поддерживает возможности холста, можно использовать любое резервное решение, которое вам по душе.

ЧАСТО ЗАДАВАЕМЫЙ ВОПРОС
Доступность холста

Можно ли повысить доступность холста?

Одной из ключевых тем HTML5, семантических элементов и нескольких предыдущих глав является доступность, т. е. разработка веб-страниц, которые предоставляют информацию для вспомогательных программ для пользователей с ограниченными возможностями. И сейчас, может быть, трудно поверить в то, что для одной из основных новых возможностей HTML5 не существует ни семантики, ни модели доступности вообще.

Создатели HTML5 работают над восполнением этого недостатка. Но ни у кого нет твердой уверенности в том, какое решение может быть лучшим. В одном из возможных решений предлагается создавать отдельный документ для вспомогательных устройств, который бы отображал содержимое холста другими средствами. Проблема с этим предложением заключается в том, что ответственность за синхронизацию этого содержимого с содержимым холста лежит на разработчике, а ленивые или перегруженные работой разработчики, скорее всего, передадут эту ответственность дальше, если ее реализация окажется сколь-нибудь сложной задачей. В другом подходе предлагается расширить возможность карт ссылок (существующей возможности HTML, которая разделяет изображение на несколько областей, активизируемых щелчком мыши), чтобы использовать их в качестве слоя поверх холста. Так как карта ссылок, по сути, является группой ссылок, она могла бы содержать важную информацию для считывания вспомогательными программами и донесения ее к пользователю.

Но в настоящее время нет смысла уделять большое внимание любой из этих идей, т. к. они обе все еще обсуждаются. Тем временем имеет смысл использовать холст для разных графических целей, таких как аркадные игры (большинство из них практически нельзя сделать доступными) и визуализации данных (при условии наличия этих данных в текстовом формате в другом месте страницы). Но холст нельзя назвать удачным выбором для универсального элемента дизайна страницы. Поэтому, если вы планируете использовать его для создания вычурного заголовка или меню для своего веб-сайта, лучше пока повременить с этим.

7

ГЛАВА 7

Продвинутые методы работы с холстом

Холст предоставляет огромный диапазон разнообразных возможностей. В предыдущей главе мы научились рисовать штриховые рисунки, и даже создали приличную программу для рисования, применив всего лишь несколько десятков строк кода JavaScript. Но это далеко не все возможности холста. С его помощью можно не только показывать динамические изображения и создавать программы рисования, но и проигрывать анимации, обрабатывать изображения с точностью до пиксела, а также играть в интерактивные игры. В этой главе мы заложим фундаментальную базу для реализации всех этих задач.

Мы начнем с рассмотрения методов контекста, позволяющих рисовать разные типы содержимого на холсте, включая изображения и текст. Далее мы научимся украшать наши рисунки тенями, узорными и радиальными заливками. Наконец, мы овладеем практическими методами для придания холсту интерактивности и проигрывания анимаций. Сможем мы создать все эти приложения, используя лишь обычный JavaScript-код и неудержимые амбиции.

Примечание

Сначала в этой главе мы будем работать с небольшими фрагментами кода для рисования. Этот код можно вставлять в свои страницы, но при этом нужно добавить элемент `<canvas>` и создать контекст рисования, как описано в *разд. "Базовые возможности холста" главы 6*. А затем мы возьмемся за более амбициозные проекты. Хотя мы сможем увидеть большинство (если не весь) кода для рисования на холсте, используемого в этих примерах, мы не сможем рассмотреть все подробности всех страниц. Для этого, и чтобы попробовать уже готовые примеры, посетите сайт книги по адресу **http://www.prosetech.com/html5/**.

Что еще можно рисовать на холсте?

На холсте можно в совершенстве нарисовать любую желаемую графику, от набора линий или простых геометрических фигур до портрета со всеми мельчайшими подробностями. Но с повышением уровня сложности графики повышается уровень сложности кода. В высшей степени маловероятно, что можно было бы написать

самостоятельно весь код, требуемый для создания высококачественного изображения.

К счастью, у разработчиков есть другие варианты, кроме как писать весь код самостоятельно. Возможности контекста рисования не ограничиваются рисованием простых прямых и кривых линий, также существуют методы для вставки готовых изображений, текста, узоров и даже рамок для показа видео. В следующих разделах мы узнаем, как использовать эти методы, чтобы обогатить содержимое нашего холста.

Вставка в холст изображений

Вам, наверное, приходилось видеть веб-страницы со спутниковыми картами того или иного региона планеты. Эти карты создаются из снимков отдельных участков поверхности земли, которые эти страницы загружают со спутника и объединяют в одно изображение. Это пример того, как можно взять несколько изображений и объединить их в одно требуемым образом.

Холст поддерживает работу с обычными изображениями посредством метода контекста рисования `drawImage()`. Чтобы вставить изображение в холст, методу `drawImage()` в параметрах передается объект изображения и координаты холста, по которым это изображение следует вставить:

```
context.drawImage(img, 10, 10);
```

Но чтобы передать объект изображения, его сначала нужно получить. В HTML5 есть три разных способа получения объекта изображения. Первый подход — создать его самостоятельно попиксельно с помощью метода `createImageData()`. Но этот подход очень трудоемкий и медленный. (Манипуляция пикселами рассматривается в *разд. "Проверка на столкновение с использованием цвета пикселов" далее в этой главе*.)

Второй подход — использовать уже имеющийся в разметке элемент `<img>`. Например, если у нас есть следующая разметка:

```
<img id="arrow_left" src="arrow_left.png">
```

изображение можно вставить в холст с помощью этого кода:

```
var img = document.getElementById("arrow_left");
context.drawImage(img, 10, 10);
```

Наконец, можно создать объект изображения и загрузить изображение из отдельного файла. Недостаток этого подхода состоит в том, что изображение нельзя использовать с методом `drawImage()` до тех пор, пока оно полностью не загрузится. Во избежание проблем нужно подождать, пока не выполнится событие изображения `onLoad`, прежде чем пытаться выполнять какие-либо операции с ним.

Чтобы разобраться в этом процессе, рассмотрим пример. Допустим, нам нужно отобразить на холсте изображение maze.png. Теоретически, это можно сделать такой последовательностью операций:

```
// Создаем объект изображения.
var img = new Image();
// Загружаем файл изображения.
img.src = "maze.png";
// Прорисовываем изображение. (Этот шаг может не выполниться,
// т. к. изображение, может быть, еще не загрузилось.)
context.drawImage(img, 0, 0);
```

Здесь проблема состоит в том, что установка атрибута `src` начинает загрузку изображения, но код продолжает исполняться дальше, не ожидая завершения загрузки. Правильно будет использовать следующий код:

```
// Создаем объект изображения.
var img = new Image();

// Привязываем функцию к событию onload
// Это указывает браузеру, что делать, когда изображение загружено.
img.onload = function() {
  context.drawImage(img, 0, 0);
};

// Загружаем файл изображения.
img.src = "maze.png";
```

Этот подход может показаться нелогичным, т. к. порядок указания операций в коде не совпадает с порядком их исполнения. В данном примере вызов метода `context.drawImage()` происходит в последнюю очередь, вскоре после установки свойства `img.src`.

Изображения имеют широкий диапазон применений. Их можно использовать, чтобы приукрасить штриховые рисунки или как альтернативу рисованию изображений самому. Изображения можно использовать для разных объектов и персонажей, разместив их соответствующим образом на холсте. Изображения можно также применять для заполнения линий, чтобы придать им текстурированный вид. Мы рассмотрим несколько практических примеров использования рисунков далее в этой главе.

АВАРИЙНАЯ СИТУАЦИЯ

Все мои рисунки искажены!

Если обнаружится, что ваше изображение по непонятной причине растянуто, сжато или искажено каким-либо другим образом, наиболее вероятной причиной этому будет установка размера холста посредством правила таблицы стилей.

Правильно устанавливать размер холста надо через указание его высоты и ширины в атрибутах `height` и `width` элемента `<canvas>` в разметке страницы. Может показаться, что эти значения не обязательно задавать в разметке, используя такую форму тега:

```
<canvas></canvas>
```

а установить их в правиле таблицы стилей следующим образом:

```
canvas {
  height: 300px;
  width:  500px;
}
```

Но этот подход не будет работать. Проблема состоит в том, что свойства `height` и `width` CSS отличаются от одноименных свойств элемента `<canvas>`. Если не указать размеры холста в разметке, будет установлен размер холста по умолчанию — 300×150 пикселов. Потом таблица стилей будет растягивать или сжимать холст, чтобы подогнать его к указанным в ней размерам, изменяя соответствующим образом размеры всего содержимого холста. В результате размещенные на холсте изображения будут искажены, что, несомненно, не сделает их привлекательными.

Во избежание этой проблемы всегда нужно указывать размер холста в разметке посредством атрибутов `height` и `width` элемента `<canvas>`. А если нужно изменить размер холста на основе каких-либо других критериев, значения этих атрибутов изменяются в разметке с помощью JavaScript.

Обрезка, разрезка и изменение размеров изображения

Функция `drawImage()` принимает несколько необязательных параметров, которые позволяют манипулировать изображением на холсте. Например, размер изображения можно изменить, указав параметры `width` и `height` следующим образом:

```
context.drawImage(img, 10, 10, 30, 30);
```

В данном случае метод размещает изображение в рамке размером 30×30 пикселов, левый верхний угол которой находится в точке холста с координатами (10, 10). Если первоначальный размер изображения был 60×60 пикселов, то эта операция уменьшает его размеры наполовину в обоих направлениях, в результате чего общий размер конечного изображения будет лишь в четверть размера исходного.

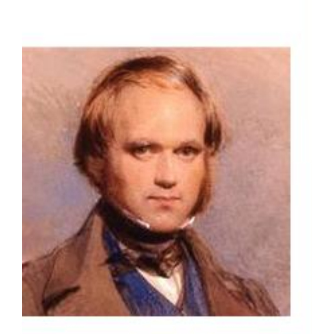
Если нужно вставить в холст только часть изображения, методу `drawImage()` необходимо передать четыре параметра в начале списка параметра. Эти параметры определяют позицию и размер части изображения, которую нужно вырезать:

```
context.drawImage(img, source_x, source_y, source_width, source_height,
                  x, y, width, height);
```

Последние четыре параметра в этом коде те же, что и в предыдущем примере — они определяют позицию и размер изображения на холсте.

Допустим, что мы хотим вставить в холст только верхнюю половину изображения с исходным размером 200×200 пикселов. Для этого с левого верхнего угла изображения (точка (0, 0)) отрезаем часть изображения шириной в 200 и высотой в 100 пикселов, которую потом вставляем в холст в начальной точке с координатами (75, 25). Все это делается в одной строчке кода:

```
context.drawImage(img, 0, 0, 200, 100, 75, 25, 200, 100);
```



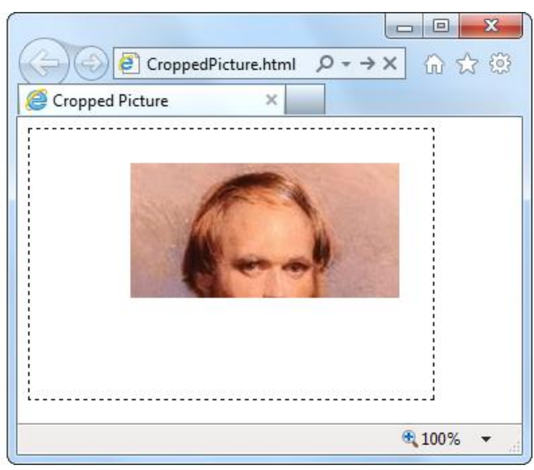


Рис. 7.1. *Слева:* исходное изображение.
Справа: обрезанная верхняя часть изображения, вставленная в холст

Исходное изображение и результаты вставки в холст его части показаны на рис. 7.1.

Возможностей метода `drawImage()` недостаточно, если вставляемое в холст изображение нужно повернуть или скосить на определенный угол. Но эти и другие манипуляции с изображением на холсте можно выполнить с помощью трансформаций, как рассматривается в *разд. "Трансформации" главы 6*.

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ
Видеокадр на холсте

В первом параметре метода `drawImage()` указывается изображение, которое нужно вставить в холст. Как мы только что увидели, таким изображением может быть созданный объект изображения или элемент `<img>`, расположенный где-то в другом месте разметки.

Но это не все, что можно вставить в холст посредством этого метода. Вместо изображения можно указать элемент `<canvas>` (но не тот, на котором выполняется настоящее рисование). Таким же образом можно вставить и элемент `<video>`, в котором воспроизводится видео:

```
var video = document.getElementById("videoPlayer");
context.drawImage(video, 0, 0, video.clientWidth, video.clientWidth);
```

При исполнении этого кода выполняется захват одного кадра видео, воспроизводимого в момент выполнения кода, который потом вставляется в холст.

Эта возможность открывает дверь для создания некоторых интересных эффектов. Например, с помощью таймера можно последовательно выполнять захват кадров воспроизводимого видео и вставлять их в холст. Если делать это достаточно быстро, то вставляемая в холст последовательность кадров будет создавать эффект проигрывающегося видео.

Но чтобы это не был просто другой видеопроигрыватель, захваченные кадры можно модифицировать перед тем, как вставлять их в холст. Например, вставляемый кадр можно увеличить или уменьшить или наложить какой-либо эффект в стиле Photoshop, модифицировав его на уровне пикселов. Один из примеров такой манипуляции см. по

адресу **http://html5doctor.com/video-canvas-magic**. В нем показано, как сделать цветное видео черно-белым, преобразуя каждый цветной пиксел захваченного кадра в оттенок серого.

Вставка в холст текста

Другим типом изображения, который нам вряд ли бы хотелось составлять из отдельных прямых и кривых, является текст. К счастью, разработчики HTML5 позаботились об этом для нас и предоставляют два метода контекста рисования для работы с текстом.

Но прежде чем вставлять в холст какой-либо текст, нужно указать шрифт для него, установив значение свойства контекста `font`. Это значение указывается посредством строки с таким же синтаксисом, как и для универсального свойства `font` CSS. Как минимум, нужно указать размер шрифта в пикселах и его название. Например, как в следующей строке кода:

```
context.font = "20px Arial";
```

В случае неуверенности в поддержке определенного шрифта браузерами, можно указать список шрифтов:

```
context.font = "20px Verdana, sans-serif";
```

Также можно установить жирный шрифт или курсив, указав соответствующие параметры в начале строки:

```
context.font = "bold 20px Arial";
```

Самое главное, благодаря CSS3 можно использовать вычурные встроенные шрифты. Для этого нужно лишь сначала зарегистрировать название шрифта в таблице стилей *(см. разд. "Типография для Интернета" главы 8)*.

После установки шрифта текст в холст вводится с помощью метода `fillText()`. Например, следующий код вводит в холст строку текста, левый верхний угол которого находится в точке (10, 10):

```
context.textBaseline = "top";
context.fillStyle = "black";
context.fillText("I'm stuck in a canvas. Someone let me out!", 10, 10);
```

Текст можно вставлять в любое место на холсте, но только по одной строке за раз. Чтобы вставить несколько строк, нужно делать соответствующее число вызовов метода `fillText()`.

Совет

Чтобы разделить абзац текста на несколько строчек, можно создать свой ***алгоритм переноса строк***. Идея заключается в следующем: предложение разбивается на слова, и с помощью метода `measureText()` определяется количество слов, которые помещаются в каждую строку. Это кропотливая работа, но на сайте **http://tinyurl.com/6ec7hld** можно найти хорошую отправную точку для ее выполнения.

Вместо метода `fillText()` можно использовать другой метод для ввода текста — `strokeText()`. Этот метод вводит очертания букв текста; цвет и толщина очертания

определяются значениями свойств контекста `strokeStyle` и `lineWidth`. Далее показан пример использования этого метода:

```
context.font = "bold 40px Verdana, sans-serif";
context.lineWidth = "1";
context.strokeStyle = "red";
context.strokeText("I'm an OUTLINE", 20, 50);
```

Как уже отмечалось, метод `strokeText()` вводит только очертания букв. Если требуется создать текст одного цвета, а его обводку — другого, можно использовать сначала метод `fillText()`, а после него метод `strokeText()`. На рис. 7.2 показан пример результатов выполнения обоих методов (по отдельности).

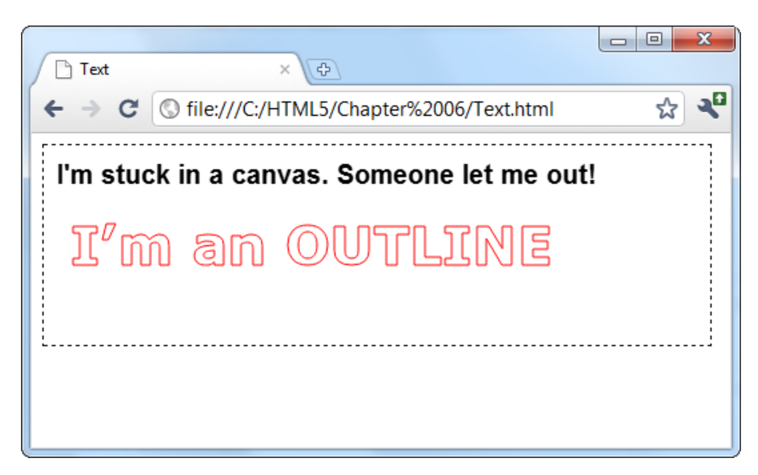


Рис. 7.2. Текст, созданный методом `fillText()` *(верхняя строчка)* и методом `strokeText()` *(нижняя строчка)*

Совет

Операция вывода текста на холст выполняется намного медленнее, чем рисование линий и даже изображений. Скорость не имеет важности при создании статического изображения (например, диаграммы), но может быть фактором при создании интерактивного приложения или анимации. Оптимизировать ввод текста можно, сначала сохранив требуемый текст в файле изображения, а потом отображая его на холсте посредством метода `drawImage()`.

Тени и вычурные заливки

До сих пор для рисования и заливки изображений на холсте мы использовали сплошные цвета. И хотя в этом определенно нет ничего предосудительного, артистические натуры будут рады узнать, что возможности холста не ограничиваются только сплошными цветами. Одним из примеров такого оформления будет возможность любого изображения на холсте отбрасывать артистически оформленную тень. Другим примером будет заполнение фигуры узором повторяющихся изображений. Но почти наверняка самой изысканной возможностью рисования являются *градиенты*, посредством которых можно смешивать два или несколько цветов, создавая калейдоскопические узоры.

В последующих разделах мы научимся использовать эти возможности, просто устанавливая разные свойства контекста рисования холста.

Создание теней

Одной из полезных возможностей холста является добавление теней позади любого нарисованного изображения. На рис. 7.3 показано несколько примеров разных типов теней.

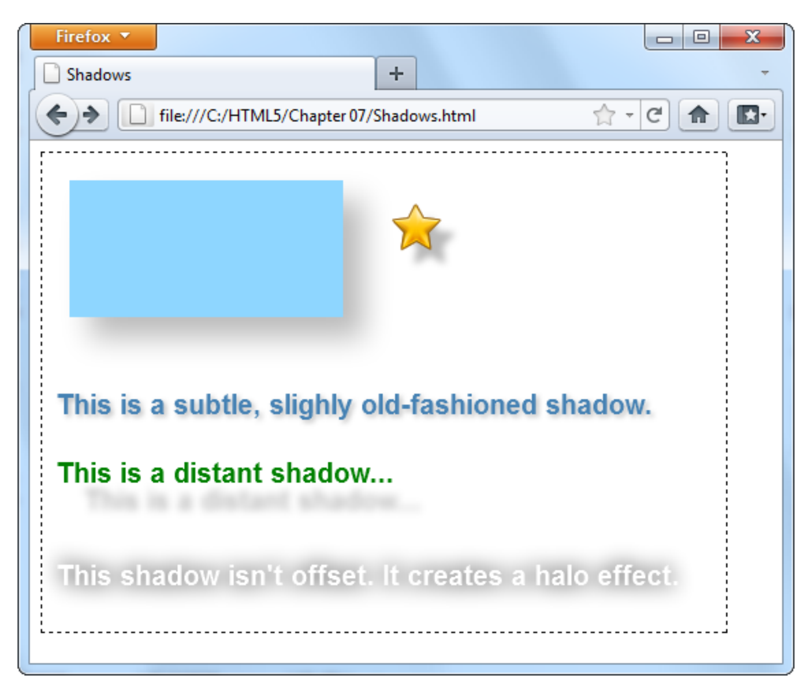


Рис. 7.3. Тени можно применять с одинаковым успехом с фигурами, изображениями и текстом. Одной из интересных особенностей является взаимодействие теней с изображениями с прозрачным фоном, такими как звезда на этом рисунке. Как можно видеть, в данном случае тень следует очертаниям звезды, а не квадрату, определяющему все изображение. (На момент написания данной книги эту возможность поддерживали только Internet Explorer и Firefox.) Тени также хорошо сочетаются с текстом, позволяя создавать разнообразные эффекты, управляемые установкой соответствующих параметров

По сути, тень — это размытая версия исходного изображения — линии, фигуры, изображения или текста. Внешний вид тени управляется установкой свойств контекста рисования (табл. 7.1).

Таблица 7.1. *Свойства для управления внешним видом теней*

Свойство	Описание
`shadowColor`	Устанавливает цвет тени. Можно установить черную или цветную тень, но обычно лучше всего делать ее полусерой. Другой хороший подход — использовать полупрозрачные тени *(см. разд. "Прозрачность" главы 6)*, чтобы можно было видеть содержимое под ними. Отключить тени можно, присвоив атрибуту альфа свойства `shadowColor` нулевое значение

Таблица 7.1 *(окончание)*

Свойство	Описание
`shadowBlur`	Устанавливает степень расплывчатости теней. Нулевое значение этого свойства определяет четкую, резкую тень, выглядящую как силуэт исходного изображения. А значение 20 дает тень в виде размытой дымки, и можно установить еще высшее значение. Большинство людей считает, что лучше всего выглядит слегка размытая тень (значение `shadowBlur` около 3)
`shadowOffsetX` и `shadowOffsetY`	Определяют положение тени относительно содержимого, которому она принадлежит. Например, если присвоить каждому свойству значение 5, тень будет расположена на 5 пикселов вправо и 5 пикселов вниз от исходного содержимого. Отрицательные значения сдвигают тень в противоположном направлении — влево и вверх

Разнообразные тени на рис. 7.3 были созданы следующим кодом:

```
// Рисуем прямоугольник с тенью.
context.rect(20, 20, 200, 100);
context.fillStyle = "#8ED6FF";
context.shadowColor = "#bbbbbb";
context.shadowBlur = 20;
context.shadowOffsetX = 15;
context.shadowOffsetY = 15;
context. fill();

// Рисуем звезду с тенью.
context.shadowOffsetX = 10;
context.shadowOffsetY = 10;
context.shadowBlur = 4;
img = document.getElementById("star");
context.drawImage(img, 250, 30);
context.textBaseline = "top";
context.font = "bold 20px Arial";

// Рисуем три строчки текста с тенью.
context.shadowBlur = 3;
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
context.fillStyle = "steelblue";
// Едва различимая, слегка старомодная тень.
context.fillText("This is a subtle, slightly old-fashioned shadow.",
                 10, 175);

context.shadowBlur = 5;
context.shadowOffsetX = 20;
context.shadowOffsetY = 20;
context.fillStyle = "green";
```

```
// Далекая тень.
context.fillText("This is a distant shadow...", 10, 225);

context.shadowBlur = 15;
context.shadowOffsetX = 0;
context.shadowOffsetY = 0;
context.shadowColor = "black";
context.fillStyle = "white";
// Эта тень не смещена от исходного изображения и создает эффект ореола.
context.fillText("This shadow isn't offset. It creates a halo effect.",
                 10, 300);
```

Заполнение фигур изображениями

Нарисованные на холсте фигуры можно заполнять не только сплошными или полупрозрачными цветами, но также градиентными цветами или вымащивать изображениями. Такие вычурные стили, несомненно, сделают простые фигуры более привлекательными. Такого рода оформление выполняется в два этапа: сначала создается заполнение, которое потом связывается со свойством `fillStyle` (или иногда со свойством `strokeStyle`).

Заполнение вымащиванием осуществляется путем множественной вставки копий одного исходного изображения вплотную друг к другу (рис. 7.4).



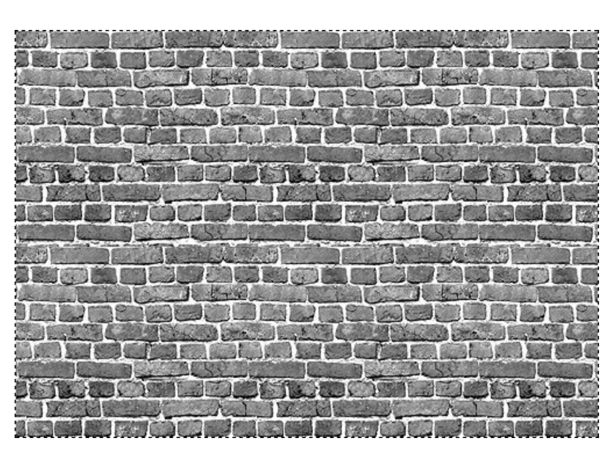


Рис. 7.4. *Слева:* исходное изображение-плитка.
Справа: область холста, заполненная плитками исходного изображения

Изображение, используемое в качестве исходной плитки, нужно загрузить в объект изображения посредством одного из методов, рассмотренных в *разд. "Вставка в холст изображений" ранее в этой главе*. В данном примере используется первый подход:

```
var img = document.getElementById("brickTile");
```

Имея объект изображения, можно создать объект шаблона, используя метод контекста `createPattern()`. На этом этапе указывается направление вымащивания —

горизонтально (`repeat-x`), вертикально (`repeat-y`) или в обоих направлениях (`repeat`).

```
var pattern = context.createPattern(img, "repeat");
```

Последний шаг — присвоить созданный объект шаблона свойству контекста `fillStyle` или `strokeStyle`:

```
context.fillStyle = pattern;
context.rect(0, 0, canvas.width, canvas.height);
context.fill();
```

Этот код создает прямоугольник, который заполняет холст исходным изображением (см. рис. 7.4).

Градиентная заливка фигур

Другим типом заполнения является градиентное, в котором смешиваются два или несколько цветов. Холст поддерживает линейные и радиальные градиенты, сравнительные примеры которых показаны на рис. 7.5.

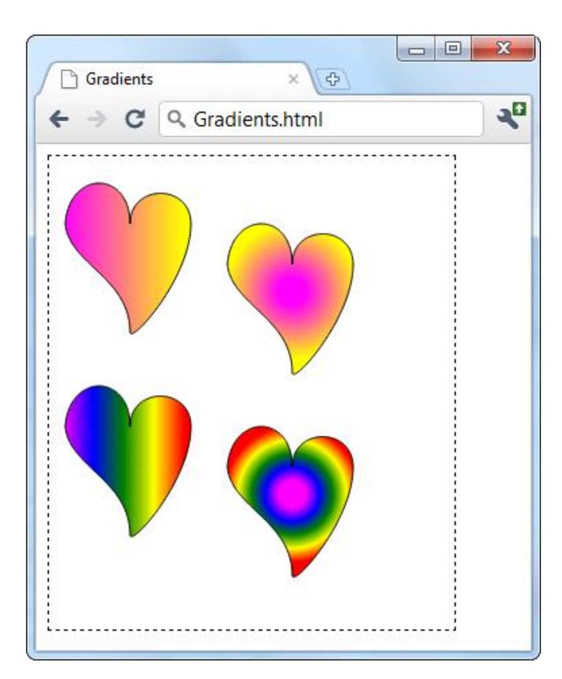


Рис. 7.5. Линейный градиент *(верхний слева)* переходит из одного цвета в другой по прямой линии. Радиальный градиент *(верхний справа)* переходит из одного цвета в другой по радиусу. Оба типа градиентов поддерживают больше, чем два цвета, позволяя создавать эффект полос или концентрических кругов, плавно переходящих из одного цвета в другой (*внизу слева* и *справа* соответственно)

Совет

Черно-белые изображения градиентов в книге вряд ли позволяют получить должное представление об этих эффектах, поэтому рекомендуется оценить их на сайте этой

книги по адресу **http://www.prosetech.com/html5/**. Там же можно ознакомиться и с кодом для создания этих эффектов, включая логику для рисования кривых Безье, из которых созданы сердцевидные фигуры.

Первым шагом в создании градиентной заливки будет создание правильного типа объекта градиента. Для решения этой задачи контекст рисования предоставляет два метода: `createLinearGradient()` и `createRadialGradient()`. Оба метода работают более-менее похоже: они содержат список цветов, которые задействуются в разных точках.

Самый легкий способ разобраться с градиентами — это изучить простой пример. В качестве такого примера рассмотрим код для создания градиента в левой верхней сердцеобразной фигуре на рис. 7.5:

```
// Создаем градиент от точки (10, 0) до точки (100, 0).
var gradient = context.createLinearGradient(10, 0, 100, 0);

// Добавляем два цвета.
gradient.addColorStop(0, "magenta");
gradient.addColorStop(1, "yellow");

// Вызываем функцию для рисования фигуры.
drawHeart(60, 50);

// Рисуем фигуру.
context.fillstyle = gradient;
context.fill();
context.stroke();
```

При создании линейного градиента указываются две точки, представляющие начало и конец пути, вдоль которого происходит изменение цвета.

Важность линии градиента состоит в том, что она определяет внешний вид градиента (рис. 7.6).

Рассмотрим, например, линейный градиент, переходящий из светло-розового цвета в желтый. Этот переход можно выполнить в полосе шириной в несколько пикселов или же по всей ширине холста. Кроме этого, направление перехода может быть слева направо, сверху вниз или же с любым наклоном между этими двумя направлениями. Все эти аспекты определяются линией градиента.

Совет

Градиент можно рассматривать как цветовой дизайн, расположенный под холстом. Эта цветовая поверхность формируется при создании градиента. При заполнении градиентом фигуры мы вырезаем отверстие по контуру этой фигуры, которое позволяет градиенту просматриваться. Конечный результат — то, что отображается на холсте — зависит как от параметров градиента, так и от размера и позиции фигуры.

В данном примере линия градиента берет начало в точке (10, 0) и оканчивается в точке (100, 0). Эти точки предоставляют нам следующую важную информацию о данном градиенте.

- **Градиент горизонтальный.** Это означает, что переход цветов происходит слева направо. Мы извлекаем эту информацию из того факта, что обе точки имеют одинаковую ординату. Если бы мы хотели выполнить переход сверху вниз, то можно было использовать, например, точки (0, 10) и (0, 100). А для перехода по диагонали сверху вниз слева направо можно было бы использовать, например, точки (10, 10) и (100, 100).
- **Собственно переход охватывает всего лишь 90 пикселов** (начиная со значения абсциссы, равного 10, и заканчивая, когда это значение равно 100). В данном примере горизонтальный размер сердцевидной фигуры слегка меньше, чем размеры градиента, вследствие чего в фигуре видно большую часть градиента.
- **Цвета за пределами этого градиента становятся сплошными.** Поэтому, если сделать фигуру шире, ее левый край будет окрашен чистым светло-розовым цветом, а правый — чистым желтым.

Совет

Часто размер градиента будет лишь слегка больше заполняемой им фигуры, как в этом примере. Но возможны и другие варианты. Например, чтобы заполнить несколько фигур разными частями градиента, можно создать градиент, покрывающий весь холст.

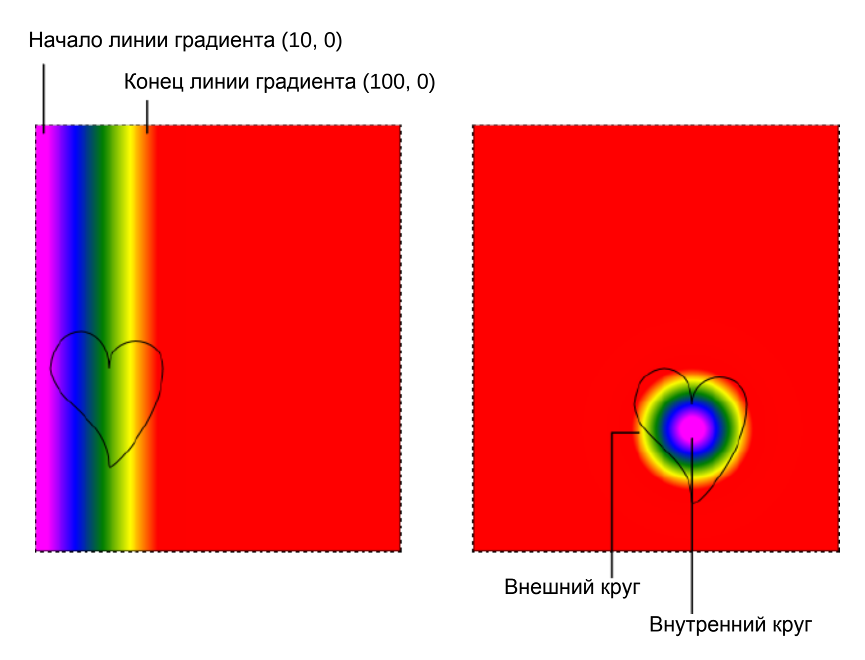


Рис. 7.6. *Слева:* градиент для нижней слева фигуры на рис. 7.5. Из градиента, используемого для заливки фигуры, отображается только часть его. *Справа:* то же справедливо и для радиального градиента, использующегося для заливки сердцевидной фигуры нижней справа фигуры на рис. 7.5

Установка цветов градиента осуществляется вызовами метода градиента `addColorStop()`. При каждом вызове метода ему передается значение смещения от `0` до `1`, которое определяет местонахождение цвета в переходе. Значение `0` означает,

что цвет находится в самом начале градиента, а значение `1` размещает цвет в самом конце. Изменив эти числа (например, на `0,2` и `0,8`), мы можем сжать градиент, показывая большую область сплошного цвета на каждом конце.

Для двухцветного градиента наиболее логичными значениями смещения будут `0` и `1`. Но для градиентов с большим количеством цветов можно устанавливать разные смещения, чтобы сжать одни цветовые полосы и растянуть другие. В левой нижней фигуре на рис. 7.5 смещения распределены равномерно, вследствие чего все цветовые полосы имеют одинаковую ширину.

```
var gradient = context.createLinearGradient(10, 0, 100, 0);
gradient.addColorStop("0", "magenta");
gradient.addColorStop(".25", "blue");
gradient.addColorStop(".50", "green");
gradient.addColorStop(".75", "yellow");
gradient.addColorStop("1.0", "red");

drawHeart(60, 200);
context.fillStyle = gradient;
context.fill();
context.stroke();
```

> **ПРИМЕЧАНИЕ**
>
> Если на данный момент вам не все понятно с градиентами, не паникуйте. В конце концов, вам не обязательно понимать все малейшие подробности процесса создания градиента, чтобы его использовать. Кроме того, вы можете поэкспериментировать со смещениями, пока не получите привлекательный переход цветов.

Процесс создания радиального градиента такой же, как и линейного, только вместо определения двух точек нужно определить два круга. Это потому, что переход цветов в радиальном градиенте распространяется с меньшего круга к большему. Эти круги определяются указанием их центра и радиуса.

В правой верхней фигуре примера радиального градиента на рис. 7.5 цветовой переход распространяется от центральной точки фигуры с координатами (180, 100). Внутренний цвет ограничен кругом радиусом 10 пикселов, а внешний — кругом радиусом 50 пикселов. Опять же, если выйти за эти пределы, мы получим сплошные цвета — светло-розовый в центре и желтый по внешней окружности.

Код для создания двухцветного радиального градиента выглядит так:

```
var gradient=context.createRadialGradient(180, 100, 10, 180, 100, 50);
gradient.addColorStop(0, "magenta");
gradient.addColorStop(1, "yellow");

drawHeart(180, 80);
context.fillStyle = gradient;
context.fill();
context.stroke();
```

Примечание

Чаще всего для внутреннего и внешнего кругов указываются одинаковые центры. Но это не обязательно, и их можно сместить относительно друг друга, что может растянуть, сжать и по-иному исказить переход, придавая ему интересный эффект.

На основе этого примера можно создать многоцветный радиальный градиент, пример которого показан справа внизу на рис. 7.5. Для этого нужно просто переместить центр кругов внутрь сердцеобразной фигуры и добавить другие цвета, те же, что и для многоцветного линейного градиента:

```
var gradient = context.createRadialGradient(180, 250, 10, 180, 250, 50);
gradient.addColorStop("0","magenta");
gradient.addColorStop(".25","blue");
gradient.addColorStop(".50","green");
gradient.addColorStop(".75","yellow");
gradient.addColorStop("1.0","red");
drawHeart(180, 230);
context.fillStyle = gradient;
context.fill();
context.stroke();
```

Теперь у вас должно быть достаточно знаний, чтобы самостоятельно создавать калейдоскопические градиенты.

Обобщая сказанное: рисуем график

На данном этапе мы разобрались с большей частью предоставляемых холстом возможностей рисования и можем, наконец, извлечь из наших знаний практическую пользу. В следующем примере мы возьмем скучный текст и цифры и представим их на холсте в виде простого, но интересного графика.

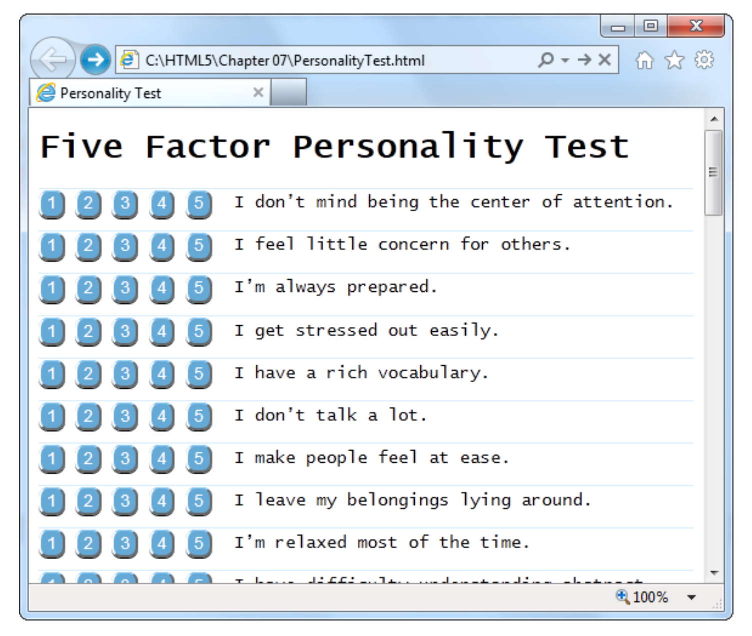
Отправная точка для этого примера показана на рис. 7.7 — личностный тест на двух страницах с минимальной графикой. Пользователь отвечает на вопросы на первой странице, а потом нажимает кнопку **Get Score**, чтобы вывести на второй странице результаты теста, полученные посредством обработки данных с первой страницы в соответствии с пресловутой пятифакторной моделью личности *(см. врезку "На профессиональном уровне. Как охарактеризовать личность по пяти числам" далее в этой главе)*.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Как охарактеризовать личность по пяти числам

В пятифакторном личностном тесте определяется тип личности на основе пяти личностных качеств: *откровенность* (openness), *сознательность* (conscientiousness), *экстраверсия* (extraversion), *податливость* (agreeableness) и *невротицизм* (neuroticism). Эти факторы были выведены на основе анализа психологами тысяч прилагательных, описывающих личность, который был выполнен посредством статистической обработки на компьютере опросов по установлению личностных качеств. Были установлены прилагательные, которые люди обычно используют совместно в ответах на вопросы, на основе чего был определен наименьший набор основных личностных качеств. Например, люди, считающие себя *дружелюбными*, также описывают себя *общительны-*

ми и *компанейскими*, поэтому можно объединить все эти качества в один личностный фактор, который психологи называют *экстраверсией*. В результате такого анализа исследователи свели почти 20 тысяч прилагательных к пяти тесно взаимосвязанным факторам.



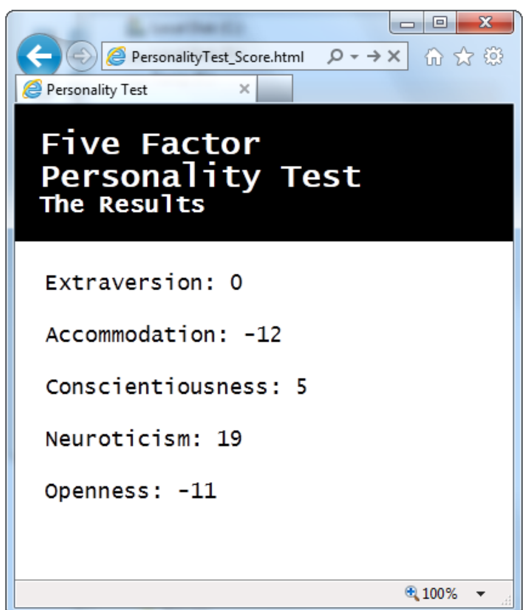


Рис. 7.7. Выберите ответы на вопросы на первой странице *(вверху)*, а потом узнайте свои результаты на второй странице *(внизу)*. К сожалению, без какого-либо визуального ориентира пользователям трудно разобраться, что эти цифры означают

Код JavaScript этого примера линейный. Когда пользователь нажимает кнопку с номером ответа, ее цвет меняется, чтобы показать сделанный выбор. Когда пользователь завершает опрос, его ответы обрабатываются простым алгоритмом по нескольким оценочным формулам, чтобы вычислить числовые значения пяти личностных факторов. Исследовать весь код или выполнить тест можно на сайте книги по адресу **http://http://www.prosetech.com/html5/**.

Пока что здесь нет никакого волшебства HTML5. Но представьте себе, как можно было бы улучшить страницу вывода результатов, если бы вместо представления в виде цифр их можно было отобразить в виде графика. На рис. 7.8 показан один из возможных вариантов такого графика.

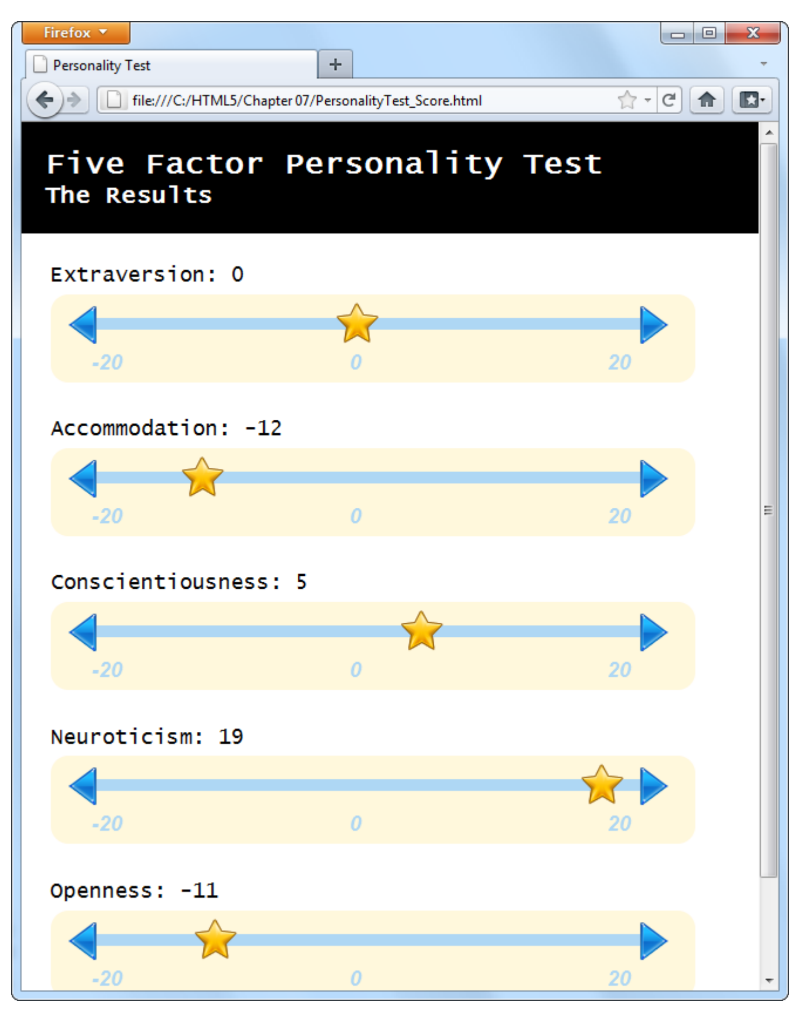


Рис. 7.8. Эта страница создана с помощью нескольких разных способов рисования на холсте, включая рисование линий, изображений и ввода текста. Но самым интересным является метод динамического рисования этих простых графиков на основе предоставляемых пользователем ответов на странице вопросов

В нем для вывода категорий результатов теста используются пять элементов холста, по одному для каждого личностного фактора. Код для создания этого графика таков:

```
<hgroup>
  <h1>Five Factor Personality Test</h1>
  <h2>The Results</h2>
</hgroup>

<div class="score">
  <h2 id="headingE">Extraversion: </h2>
  <canvas id="canvasE" height="75" width = "550"></canvas>
</div>

<div class="score">
  <h2 id="headingA"Accommodation: </h2>
  <canvas id="canvasA" height="75" width = "550"></canvas>
</div>

<div class="score">
  <h2 id="headingC">Conscientiousness: </h2>
  <canvas id="canvasC" height="75" width = "550"></canvas>
</div>

<div class="score">
  <h2 id="headingN">Neuroticism: </h2>
  <canvas id="canvasN" height="75" width="550"></canvas>
</div>

<div class="score">
  <h2 id="headingO">Openness: </h2>
  <canvas id="canvasP" height="75" width="550"></canvas>
</div>
```

Все графики рисуются одной и той же специальной функцией JavaScript, называющейся `plotScore()`. Она вызывается пять раз, каждый раз с разными параметрами. Например, чтобы нарисовать график для фактора экстраверсии вверху страницы, код передает функции самый верхний элемент холста, оценку экстраверсии (как число от –20 до 20) и название фактора ("Extraversion"):

```
window.onload = function() {
  ...
  // Получаем элемент <canvas> для оценки фактора экстраверсии.
  var canvasE = document.getElementById("canvasE");
  // Добавляем полученную оценку в соответствующее название фактора.
  // (Оценка хранится в переменной extraversion.)
  document.getElementById("headingE").innerHTML += extraversion;
```

```
  // Прорисовываем оценку в соответствующем элементе холста.
  plotScore(canvasE, extraversion, "Extraversion");
  ...
}
```

Функция `plotScore()` содержит внушительный объем кода для рисования, который к этому времени должен быть вам знакомым. Она использует разные методы контекста рисования для отображения отдельных составляющих графика:

```
function plotScore(canvas, score, title) {
  var context = canvas.getContext("2d");

  // Рисуем стрелки на концах линии графика.
  var img = document.getElementById("arrow_left");
  context.drawImage(img, 12, 10);
  img = document.getElementById("arrow_right");
  context.drawImage(img, 498, 10);

  // Рисуем линию между стрелками.
  context.moveTo(39, 25);
  context.lineTo(503, 25);
  context.lineWidth = 10;
  context.strokeStyle = "rgb(174,215,244)";
  context.stroke();

  // Отображаем числа на шкале графика.
  context.fillStyle = context.strokeStyle;
  context.font = "italic bold 18px Arial";
  context.textBaseline = 'top';
  context.fillText("-20", 35, 50);
  context.fillText("0", 255, 50);
  context.fillText("20", 475, 50);

  // Отображаем звездочку, указывающую полученную оценку
  // на шкале графика.
  img = document.getElementById("star");
  context.drawImage(img, (score+20)/40*440+35-17, 0);
}
```

Самой важной является последняя строка кода, которая вставляет звездочку в соответствующем месте шкалы с помощью следующего, слегка неопрятного, уравнения:

```
context.drawImage(img, (score+20)/40*440+35-17, 0);
```

Логика этого уравнения следующая. Первый шаг — это преобразование оценки в проценты от 0 до 100. Так как числовое значение оценки находится в диапазоне значений от –20 до 20, первой выполняемой кодом операцией будет изменение его на значение в диапазоне от 0 до 40:

```
score+20
```

А чтобы получить процентное выражение оценки, откорректированное число делится на 40:

```
(score+20)/40
```

Полученные проценты нужно умножить на длину линии. Таким образом, 0% будет в самом левом конце шкалы, 100% — на противоположном, а все остальные результаты — где-то между этими двумя:

```
score+20)/40*440
```

Этот код работал бы как следует, если бы концы линии шкалы находились в точках с абсциссами 0 и 400. Но в действительности шкала немного смещена от левого края окна, чтобы она не сливалась с ним. Поэтому звездочку необходимо сместить вправо на такое же расстояние:

```
(score+20)/40*440+35
```

Но таким образом с правильной позицией на шкале совмещается левый край звездочки, в то время как мы хотим совмещать ее середину. Эта проблема решается путем вычитания из предыдущего результата приблизительно половины ширины звездочки:

```
(score+20)/40*440+35-17
```

Это, наконец, дает нам конечное значение абсциссы положения звездочки на шкале в соответствии с оценкой данного фактора.

Примечание

Всего лишь небольшой шаг отделяет статические рисунки от динамической графики, изменяющейся в зависимости от предоставленных данных, наподобие той, которая была использована в этом примере. Но, сделав этот шаг, вы сможете применять свои знания и навыки для создания разнообразной управляемой данными графики, от традиционных секторных диаграмм до инфографики с использованием шкал и счетчиков. Если вы хотите облегчить себе эту задачу, стоит подумать о применении графической библиотеки, содержащей готовые процедуры JavaScript для рисования распространенных типов графиков на основе предоставляемых данных. В качестве двух хороших примеров такой библиотеки можно назвать Rgraph (**www.rgraph.net**) и ZingChart (**www.zingchart.com**).

Как сделать фигуры интерактивными?

Холст является *незапоминаемой* (non-retained) поверхностью рисования. Это означает, что холст не отслеживает выполняемые на нем операции рисования, он фиксирует лишь конечный результат этих операций — набор разноцветных пикселов, составляющих его содержимое.

Например, если нарисовать красный квадрат в центре холста методом `stroke()` или `fill()`, как только завершится выполнение этого метода, квадрат станет ничем иным, как блоком красных пикселов. Он может казаться вам квадратом, но холст не обладает информацией об этом.

Такой подход позволяет сделать процесс рисования быстрым, но он также усложняет задачу создания интерактивной графики. Допустим, что нам нужно разработать более интеллектуальную версию программы рисования, которую мы рассмотрели в *разд. "Создание простой программы рисования" главы 6*. В частности, мы хотим, чтобы кроме линий в ней также можно было бы рисовать прямоугольники. (Но это еще легко!) Более того, мы хотим не только рисовать прямоугольники, но также и выбирать нарисованные прямоугольники, перетаскивать их на новое место, изменять их размеры, цвет и т. п. Но для того чтобы реализовать все эти возможности, нам нужно решить несколько проблем. Прежде всего, как нам узнать, что пользователь щелкнул на прямоугольнике? Потом, как нам узнать все подробности о данном прямоугольнике — его координаты, размер, цвет контура и заполнения? Наконец, как нам выяснить подробности обо всех *прочих* фигурах на холсте? Что нам необходимо знать, если мы хотим изменить прямоугольник и обновить холст?

Чтобы решить все эти проблемы и сделать холст интерактивным, нам нужно отслеживать все рисуемые на нем объекты. Потом, когда пользователь щелкнет в какой-либо точке холста, нужно определить, не попал ли он по одной из фигур. Этот процесс называется *проверкой на столкновение* (hit testing). Если мы можем решить эти две задачи, решение остальных — модифицирование фигуры и обновление холста соответствующим образом — будет легким.

Отслеживание нарисованного содержимого

Для того чтобы изменять и обновлять содержимое холста, нам необходимо иметь всю информацию об этом содержимом. Возьмем, например, интерактивную программу для рисования кругов, окно которой показано на рис. 7.9.

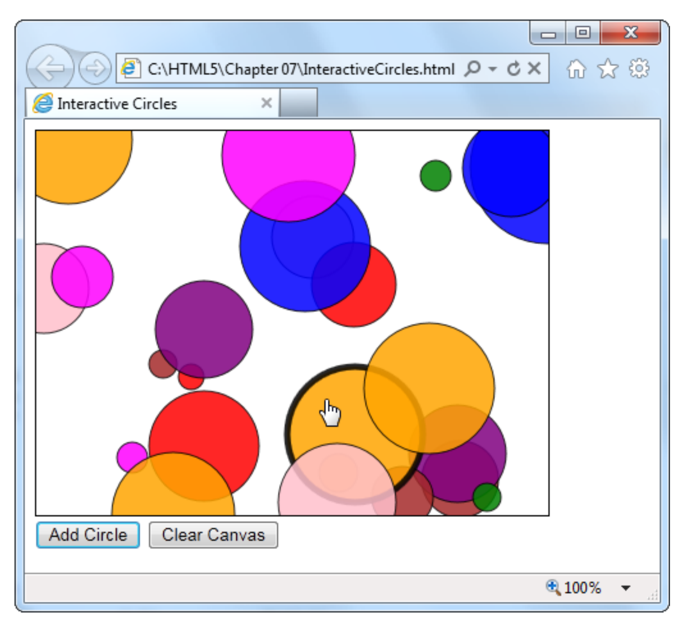


Рис. 7.9. Интерактивная программа, рисующая круги. Можно выбрать какой-либо круг, щелкнув по нему мышью (у выбранного круга появляется более толстый контур), и перетащить его на новое место

Для простоты, программа рисует только отдельные круги разного размера и цвета. Чтобы отслеживать отдельный круг, нам нужно знать его позицию на холсте (т. е. координаты центра), радиус и цвет заливки. Вместо того чтобы создавать несколько десятков переменных для хранения всей этой информации, нужно хранить эти четыре типа данных в одном компактном пакете. Таким пакетом будет *пользовательский объект* (custom object).

Рассмотрим создание пользовательского объекта на случай, если вам никогда раньше не приходилось выполнять эту стандартную процедуру. Сначала создается функция с названием, отображающим тип нашего объекта. Например, функцию для создания пользовательского объекта для рисования круга можно назвать `Circle()`:

```
function Circle() {
}
```

Мы хотим, чтобы наш объект мог хранить данные. Это делается посредством создания свойств с помощью ключевого слова `this`. Например, чтобы определить свойство `radius` объекта круга, присваивается значение выражению `this.radius`.

Далее приводится пример кода функции для создания кругов, которая сохраняет три типа информации: абсциссу, ординату и радиус круга.

```
function Circle() {
  this.x = 0;
  this.y = 0;
  this.radius = 15;
}
```

Теперь с помощью функции `Circle()` можно создавать новые объекты круга. Но здесь есть еще одна особенность — мы не хотим вызывать для этого нашу функцию, а вместо этого создавать новую копию объекта с помощью ключевого слова `new`, как показано в этом примере:

```
// Создаем новый объект круга circle и сохраняем
// его в переменной myCircle.
var myCircle = new Circle();
```

Все данные этого круга теперь доступны в виде свойств:

```
// Изменяем радиус.
myCircle.radius = 20;
```

Можно пойти далее и передавать параметры функции `Circle()`. Таким образом, создание объекта круга и установка его свойств осуществляется в одном шаге. В следующем коде приводится пример функции `Circle()`, позволяющей устанавливать параметры. (Эта функция использовалась для создания кругов для примера, показанного на рис. 7.9.)

```
function Circle(x, y, radius, color) {
  this.x = x;
  this.y = y;
  this.radius = radius;
```

```
  this.color = color;
  this.isSelected = false;
}
```

Свойство `isSelected` принимает значения `true` или `false`. Когда пользователь щелкает на круге, свойству `isSelected` присваивается значение `true`, вследствие чего код рисования знает, что у данного круга нужно нарисовать другой контур.

Объект круга с помощью этой версии функции `Circle()` можно создать посредством такого кода:

```
var myCircle = new Circle(0, 0, 20, "red");
```

Но вся идея создания программы рисования кругов заключается в том, чтобы можно было рисовать не один круг, а сколько угодно. Это означает, что только одного объекта круга будет недостаточно, и нам нужно создать массив для хранения всех кругов. В данном примере эту задачу выполняет следующая глобальная переменная:

```
var circles = [];
```

Оставшийся код не представляет ничего сложного. Когда пользователь нажимает кнопку **Add Circle**, чтобы создать новый круг, вызывается функция `addRandomCircle()`, которая создает новый круг произвольного размера и цвета в произвольном месте холста:

```
function addRandomCircle() {
  // Устанавливаем произвольный размер и позицию круга.
  var radius = randomFromTo(10, 60);
  var x = randomFromTo(0, canvas.width);
  var y = randomFromTo(0, canvas.height);

  // Окрашиваем круг произвольным цветом.
  var colors = ["green", "blue", "red", "yellow", "magenta",
                "orange", "brown", "purple", "pink"];
  var color = colors[randomFromTo(0, 8)];

  // Создаем новый круг.
  var circle = new Circle(x, y, radius, color);

  // Сохраняем его в массиве.
  circles.push(circle);

  // Обновляем отображение круга.
  drawCircles();
}
```

В этом коде применяется пользовательская функция `randomFromTo()`, которая генерирует произвольные числа в заданном диапазоне. (С полным кодом этой функции можно ознакомиться на сайте книги по адресу **http://www.prosetech.com/html5/**.)

Последним шагом в этой последовательности будет собственно прорисовка текущей коллекции кругов на холсте. Для этого после создания нового круга функция `addRandomCircle()` вызывает функцию `drawCircles()`, которая в свою очередь перебирает массив кругов с помощью следующего кода цикла:

```
for(var i=0; i<circles.length; i++) {
  var circle = circles[i];
  ...
}
```

В коде используется цикл типа `for` *(см. разд. "Циклы" приложения 2)*. Блок кода в фигурных скобках исполняется для каждого круга в массиве. Чтобы с текущим кругом было удобно работать, первая строка кода сохраняет его в переменной.

В следующем листинге приводится полный код функции `drawCircles()`, которая прорисовывает на холсте текущую коллекцию кругов:

```
function drawCircles() {
  // Очищаем холст и подготавливаемся к рисованию.
  context.clearRect(0, 0, canvas.width, canvas.height);
  context.globalAlpha = 0.85;
  context.strokeStyle = "black";

  // Перебираем все круги.
  for(var i=0; i<circles.length; i++) {
    var circle = circles[i];

    // Рисуем текущий круг.
    context.beginPath();
    context.arc(circle.x, circle.y, circle.radius, 0, Math.PI*2);
    context.fillStyle = circle.color;
    context.fill();
    context.stroke();
  }
}
```

Примечание

При каждом обновлении холста программа рисования сначала полностью очищает холст посредством метода `clearRect()`. Параноидные программисты могут удариться в панику, что эта операция может вызвать мигание холста, когда все круги сначала исчезают с него, а потом отображаются снова в обновленном состоянии. Но их беспокойство напрасно, т. к. холст оптимизирован с целью предотвратить эту проблему. В действительности на холсте ничего не очищается и не рисуется до тех пор, пока программа рисования не закончит свою работу, после чего она копирует конечный результат на холст одним плавным движением.

Нарисованные нами круги еще не обладают интерактивностью, но страница уже содержит крайне важную структуру для отслеживания каждого нарисованного круга. То есть, хотя холст все еще является просто блоком цветных пикселов, код знает точное местонахождение набора кругов, содержащихся в холсте, а это означает, что он может манипулировать этими кругами в любое время.

В следующем разделе мы рассмотрим, как использовать эту систему, чтобы разрешить пользователю выбирать круг.

Проверка на столкновение посредством сравнения координат

При работе с интерактивными фигурами почти наверняка нужно выполнять проверку на столкновение, т. е. "столкнулась" ли данная точка одной фигуры с другой фигурой. В нашей программе рисования кругов нам нужно проверить, столкнулась ли точка, в которой щелкнули мышью, с кругом или же с пустым пространством холста.

Сложные системы анимации (такие как предоставляемые Flash и Silverlight) облегчают работу разработчика и сами выполняют проверку на столкновение. Также существуют библиотеки расширения JavaScript для холста (например, Kinetic JS), направленные на предоставление таких удобств, но на момент написания данной книги ни одна из них не являлась достаточно развитой, чтобы ее можно было порекомендовать для применения. Поэтому на данное время фанатикам холста нужно разрабатывать собственную логику проверки на столкновение.

При выполнении такой проверки для каждой фигуры нам нужно вычислить, не находится ли проверяемая точка внутри пределов этой фигуры. Если находится, то точка, по которой щелкнули мышью, "столкнулась" с этой фигурой. Концептуально, это линейный процесс, но подробности его реализации — вычисления для определения, щелкнули ли по данной фигуре — могут принять довольно неуклюжие формы.

Первое, что нам требуется, — это цикл для перебора всех фигур. Этот цикл несколько отличается от цикла, используемого в функции `drawCircles()`:

```
for (var i=circles.length-1; i>=0; i--) {
  var circle = circles[i];
  ...
}
```

Обратите внимание, что этот код перебирает объекты в массиве в обратном порядке — от конца к началу. Проверка начинается с конечного элемента массива (индекс которого равен общему числу объектов в массиве минус единица) и ведет отсчет в обратном направлении к первому элементу (индекс которого равен 0). Для такого обратного направления цикла имеется причина. В частности, в большинстве приложений (включая и нашу программу рисования кругов) объекты отображаются на холсте в том порядке, в котором они хранятся в массиве. Это означает, что нарисованные позже объекты накладываются на более ранние, а при накладывании объектов щелчок получает тот объект, который находится сверху.

Чтобы определить, попал ли щелчок в фигуру, нам нужно применить математику определенного уровня. В случае круга нам нужно вычислить расстояние по прямой линии от точки, в которой щелкнули, до центра круга. Если это расстояние меньше или равно радиусу круга, тогда эта точка находится в пределах круга.

В данном примере веб-страница обрабатывает событие холста `onClick`, чтобы проверить столкновение точки, в которой щелкнули, с кругом. Щелчок по холсту активизирует функцию `canvasClick()`, которая вычисляет координаты точки, в которой щелкнули, а потом проверяет, не находятся ли они внутри какого-либо круга:

```
function canvasClick(e) {
  // Получаем координаты точки холста, в которой щелкнули.
  var clickX = e.pageX — canvas.offsetLeft;
  var clickY = e.pageY — canvas.offsetTop;

  // Проверяем, щелкнули ли по кругу.
  for (var i=circles.length; i>0; i--) {
    // С помощью теоремы Пифагора вычисляем расстояние от
    // точки, в которой щелкнули, до центра текущего круга.
    var distanceFromCenter = Math.sqrt(Math.pow(circle.x — clickX, 2) +
                                       Math.pow(circle.y — clickY, 2))

    // Определяем, находится ли точка, в которой щелкнули, в данном круге.
    if (distanceFromCenter <= circle.radius) {
      // Сбрасываем предыдущий выбранный круг.
      if (previousSelectedCircle != null) {
          previousSelectedCircle.isSelected = false;
      }
      previousSelectedCircle = circle;

      // Устанавливаем новый выбранный круг.
      circle.isSelected = true;
      // Update the display.
      drawCircles();

      // Прекращаем проверку.
      return;
    }
  }
}
```

> ***Примечание***
>
> Мы рассмотрим другой способ проверки на столкновение — через цвет пикселов — при создании игры "Лабиринт" в *разд. "Проверка на столкновение с использованием цвета пикселов" далее в этой главе*.

Для завершения этого примера код рисования функции `drawCircles()` требует небольшой доводки. Теперь он должен обработать выбранный круг особым образом, т. е. нарисовать у него жирный контур:

```
function drawCircles() {
...
  // Перебираем все круги.
  for(var i=0; i<circles.length; i++) {
      var circle = circles[i];
```

```
        if (circle.isSelected) {
            context.lineWidth = 5;
        }
        else {
            context.lineWidth = 1;
        }
        ...
    }
}
```

Этот пример можно улучшить множеством разнообразных способов и сделать его более интеллектуальным. Например, можно добавить панель инструментов с командами для редактирования выбранного круга, чтобы изменить его цвет или удалить с холста. Или же можно добавить возможность перетаскивания выбранного круга с одного места холста в другое. Для этого нужно просто отслеживать холст на событие `onMouseMove`, изменить соответствующим образом координаты круга, а потом вызвать функцию `drawCircles()`, чтобы обновить холст с кругом в новом месте. (Этот метод является, по сути, вариантом логики обработки событий мыши в программе рисования, рассмотренной в *разд. "Рисование на холсте" главы 6*, с той разницей, что движения мыши используются не для рисования линии, а для перемещения фигуры.) На веб-сайте книги (см. **http://www.prosetech/html5**) можно ознакомиться с кодом этого варианта примера (находится в файле InteractiveCircles_WithDrag.html), а также испытать его в действии.

Подводя итоги, можно сказать следующее: отслеживание рисуемых объектов предоставляет разработчику неограниченную гибкость для их редактирования и отображения в дальнейшем.

Анимация на холсте

Нарисовать одну картинку может казаться достаточно обескураживающей задачей. Поэтому не удивительно, что даже закаленные веб-разработчики рассматривают с определенным трепетом идею рисования нескольких десятков изображений каждую секунду. Основная трудность с анимацией состоит в необходимости обновлять отображение содержимого холста достаточно быстро, чтобы оно выглядело натурально движущимся или изменяющимся.

Анимация — это очевидная и необходимая составная часть для определенных типов приложений, таких как игры в режиме реального времени и эмуляторы физических процессов. Но для намного более широкого диапазона страниц на основе холста лучше применять более простые типы анимации. Так, анимацию можно использовать для выделения действий пользователя, например, подсвечивать фигуру при наведении на нее курсора мыши или заставлять ее пульсировать или мерцать. Анимацию также можно использовать для привлечения внимания к изменениям содержимого, например, постепенно вводить новый вид или создавать графики и диаграммы, "вырастающие" в требуемую позицию. Такие способы использования анимации являются мощным средством для придания глянца веб-приложениям,

уменьшения времени их отзыва и даже их выделению на фоне множества интернет-конкурентов.

Простая анимация

Сделать анимацию из рисунка на холсте HTML5 достаточно просто. Для этого устанавливается таймер, который постоянно вызывает рисунок, обычно 30 или 40 раз в секунду. При каждом вызове код полностью обновляет содержимое всего холста. Если код написан правильно, постоянно сменяющиеся кадры сольются в плавную, реалистичную анимацию.

JavaScript предоставляет два способа для управления этим повторяющимся обновлением содержимого холста.

- **Функция `setTimeout()`.** Эта функция дает указание браузеру подождать несколько миллисекунд, а потом исполнить фрагмент кода, в данном случае код для обновления содержимого холста. По окончании исполнения кода функция `setTimeout()` выполняется опять, вновь вызывая код обновления холста, и так до тех пор, пока анимацию не нужно завершить.
- **Функция `setInterval()`.** Эта функция дает указание браузеру исполнять фрагмент кода через регулярный интервал времени, например каждые 20 мс. Эффект от этой функции, по большому счету, такой же, как и от функции `setTimeout()`, но функцию `setInterval()` нужно вызывать только один раз. Чтобы остановить повторяющийся вызов кода браузером, исполняется функция `clearInterval()`.

Если код рисования быстрый и эффективный, обе функции дают одинаковый эффект. В противном случае функция `setInterval()` даст лучший результат, выполняя обновление точно в требуемое время, но, возможно, за счет производительности. При худшем стечении обстоятельств, когда исполнение кода рисования занимает несколько больше времени, чем установленный интервал, браузер будет напрягаться, чтобы поспевать, код рисования будет исполняться постоянно и страница может на мгновение зависать. По этой причине в примерах этой главы будет применяться подход с использованием функции `setTimeout()`.

При вызове функции `setTimeout()` передаются два параметра: название функции, которою нужно выполнить, и период времени ожидания перед исполнением этой функции. Время ожидания указывается в миллисекундах (т. е. тысячных секунды), таким образом, 20 мс (обычная задержка при анимации) равняется 0,02 с. Далее показан пример такого кода анимации с использованием функции `setTimeout()`:

```
var canvas;
var context;

window.onload = function() {
  canvas = document.getElementById("canvas");
  context = canvas.getContext("2d");

  // Обновляем холст через 0,02 секунды.
  setTimeout("drawFrame()", 20);
};
```

Этот вызов функции `setTimeout()` является ключевой частью любой задачи анимации. Допустим, что мы хотим создать анимацию квадрата, падающего сверху вниз страницы. Для этого нам нужно отслеживать позицию квадрата, используя две глобальные переменные:

```
// Устанавливаем начальную позицию квадрата.
var squarePosition_y = 0;
var squarePosition_x = 10;
```

Теперь нам надо просто изменять позицию при каждом исполнении функции `drawFunction()`, а потом перерисовывать квадрат в новой позиции:

```
function drawFrame() {
  // Очищаем холст.
  context.clearRect(0, 0, canvas.width, canvas.height);

  // Вызываем метод beginPath(), чтобы убедиться,
  // что мы не рисуем часть уже нарисованного содержимого холста.
  context.begin Path();

  // Рисуем квадрат размером 10x10 пикселов в текущей позиции.
  context.rect(squarePosition_x, squarePosition_y, 10, 10);
  context.lineStyle = "black";
  context.lineWidth = 1;
  context.stroke();

  // Перемещаем квадрат вниз на 1 пиксел (где он будет
  // прорисован в следующем кадре).
  squarePosition_y += 1;

  // Рисуем следующий кадр через 20 миллисекунд.
  setTimeout ("drawFrame()", 20);
}
```

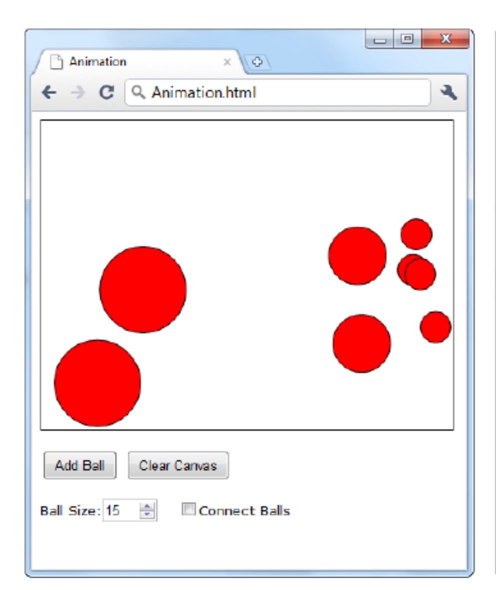
Результатом выполнения этого кода будет квадрат, подающий с верхнего края холста и исчезающий после прохода через нижний край.

Для более утонченной анимации потребуются более сложные вычисления. Например, предыдущий пример можно немного усложнить, ускоряя падение квадрата, чтобы эмулировать силу притяжения, или отскакивать от "пола", для чего потребуются более сложные математические расчеты, чем для единообразного движения вниз. Но основной подход — установка таймера, вызов функции рисования и обновление содержимого всего холста — остается точно таким же.

Анимация нескольких объектов

Теперь, когда мы освоили основы анимации и рисования интерактивной графики на холсте, настало время сделать следующий шаг и совместить эти навыки. Сделаем мы это на примере программы анимации падающих "мячиков" (рис. 7.10).

В этой программе используется уже знакомый нам по предыдущему разделу метод `setTimeout()`, но сейчас код рисования должен управлять практически неограниченным количеством падающих мячиков.



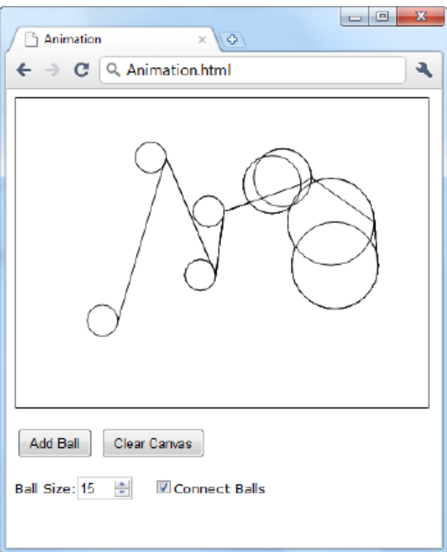


Рис. 7.10. Эта программа анимации позволяет добавлять в холст неограниченное количество мячиков. Программа предоставляет опцию выбора размера мячика (радиус по умолчанию — 15 пикселов) и соединения мячиков линиями, как показано справа на рисунке. Каждый добавленный в холст мячик двигается независимо от других, падая с ускорением, пока не столкнется с нижним краем холста, потом отскакивает от него и начинает двигаться в другом направлении

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Производительность анимации

Так как для анимации требуется постоянно обновлять нарисованную графику, очевидно, что эта задача будет намного требовательней к возможностям холста, чем обычное рисование. Но холст справляется с этой трудной задачей на удивление хорошо. В современных браузерах применяются такие средства повышения производительности, как аппаратное ускорение, при котором определенный объем обработки графики передается видеокарте компьютера, вместо применения центрального процессора для выполнения всех задач. И хотя JavaScript — не самый быстрый из языков программирования, на нем вполне можно создавать сложные высокоскоростные анимации, включая аркадные игры в режиме реального времени, не используя ничего, кроме кода сценариев и холста.

Но производительность холста *может* быть проблемой в случае маломощных мобильных устройств, таких как iPhone или устройства с операционной системой Android. Результаты тестов показывают, что анимация, которая может выполняться на настольном компьютере со скоростью 60 кадр/с (кадров в секунду), будет исполняться на смартфоне рывками, с максимальной скоростью 10 кадр/с. Поэтому, если вы хотите создавать приложения для мобильных устройств, прежде чем углубляться в разработку, обязательно протестируйте их производительность для таких устройств и будьте готовы пожертвовать некоторыми второстепенными украшениями анимационной программы, чтобы ее основная функция работала без проблем.

СОВЕТ

Судить о возможностях анимации невозможно по изображению, да еще и черно-белому. Чтобы увидеть, как анимация, представленная картинкой на рис. 7.10, выглядит в действительности, испытайте ее в действии на веб-сайте этой книги по адресу **http://www.prosetech.com/html5/**.

Для управления всеми этими шариками мы воспользуемся пользовательским объектом, рассмотренным в *разд. "Отслеживание нарисованного содержимого" ранее в этой главе*. Только в данном случае нам нужно отслеживать массив объектов мячика и кроме позиции (представляемой свойствами `x` и `y`) для каждого мячика надо еще отслеживать и скорость (представляемую свойствами `dx` и `dy`):

```
// Данные, представляющие отдельный мячик.
function Ball(x, y, dx, dy, radius) {
  this.x = x;
  this.y = y;
  this.dx = dx;
  this.dy = dy;
  this.radius = radius;
  this.color = "red";
}

// Массив, содержащий информацию обо всех мячиках на холсте.
var balls = [];
```

ПРИМЕЧАНИЕ

В математике выражение *dx* обозначает скорость изменения абсциссы, а *dy* — скорость изменения ординаты. Поэтому по мере падения мячика значение `x` для каждого кадра увеличивается на величину `dx`, а значение `y` — на величину `dy`.

При нажатии кнопки **Add Ball** простой код создает новый объект мячика `ball` и сохраняет его в массиве `balls`:

```
function addBall() {
  // Устанавливаем размер мячика.
  var radius = parseFloat(document.getElementById("ballSize").value);

  // Создаем новый мячик.
  var ball = new Ball(50,50,1,1,radius);

  // Сохраняем его в массиве balls.
  balls.push(ball);
}
```

Кроме очистки холста, кнопка **Clear Canvas** также очищает массив `balls`:

```
function clearBalls() {
  // Удаляем все мячики.
  balls = [];
}
```

Но ни функция addBall(), ни функция clearBalls() в действительности не только ничего не рисует, но даже не вызывает функцию для рисования. Вместо этого код страницы устроен таким образом, чтобы вызывать функцию drawFrame(), которая прорисовывает холст каждые 20 мс:

```
var canvas;
var context;

window.onload = function() {
  canvas = document.getElementById("canvas");
  context = canvas.getContext("2d");

  // Перерисовываем холст каждые 20 миллисекунд.
  set Timeout("drawFrame()", 20);
};
```

Функция drawFrame() является ключевой частью этого примера. Она не только рисует мячики на холсте, но также и вычисляет их текущую позицию и скорость. В функции drawFrame() выполняется несколько разных вычислений, чтобы более реалистично эмулировать движение мячиков, например, ускорение мячиков при падении и замедление, когда они отскакивают от препятствий. Полный код функции выглядит так:

```
function drawFrame() {  // Очищаем холст.
  context.clearRect(0, 0, canvas.width, canvas.height);
  context.beginPath();

  // Перебираем все мячики.
  for(var i=0; i<balls.length; i++) {
    // Перемещаем каждый мячик в его новую позицию.
    var ball = balls[i];
    ball.x += ball.dx;
    ball.y += ball.dy;

    // Добавляем эффект "гравитации", который ускоряет падение мячика.
    if ((ball.y) < canvas.height) ball.dy += 0.22;

    // Добавляем эффект "трения", который замедляет боковое движение.
    ball.dx = ball.dx * 0.998;

    // Если мячик натолкнулся на край холста, отбиваем его.
    if ((ball.x + ball.radius > canvas.width) ||
        (ball.x — ball.radius < 0)) {
         ball.dx = -ball.dx;
    }

    // Если мячик упал на низ холста, отбиваем его,
    // но слегка уменьшаем скорость.
```

```
    if ((ball.y + ball.radius > canvas.height)||
        (ball.y — ball.radius < 0)) {
         ball.dy = -ball.dy*0.96;
    }

    // Проверяем, хочет ли пользователь соединительные линии.
    if (!document.getElementById("connectedBalls").checked) {
         context.begin Path();
         context.fillStyle = ball.fillColor;
    }
    else { context.fillStyle = "white"; }

    // Рисуем мячик.
    context.arc(ball.x, ball.y, ball.radius, 0, Math.PI*2);
    context.lineWidth = 1;
    context.fill();
    context.stroke();
  }

  // Рисуем следующий кадр через 20 миллисекунд.
  setTimeout("drawFrame()", 20);
}
```

Совет

Ознакомиться с описанием логических операторов можно в табл. П2.1 *(см. приложение 2)*.

Хотя весь этот объем кода может вызывать легкий страх, общий подход остался прежним. Код исполняет следующие операции:

1. Очищает холст.
2. Перебирает в цикле мячики в массиве.
3. Корректирует позицию и скорость каждого мячика.
4. Рисует каждый мячик на холсте.
5. Устанавливает время ожидания (функция `setTimeout()`) для вызова метода `drawFrame()` опять 20 мс.

Сложным здесь является *шаг 3*, в котором корректируются позиция и скорость мячиков. Уровень сложности этого кода может быть таким, каким этот код пожелает разработчик, в зависимости от эффектов, которые он пытается реализовать. Особенно трудно поддается моделированию плавное, природное движение, требующее более сложных математических вычислений.

Наконец, теперь, когда мы можем отслеживать каждый мячик, добавить интерактивность не составляет труда. По сути, можно использовать практически тот же код, который применяется для определения щелчков по кругу в программе рисования кругов *(см. разд. "Проверка на столкновение посредством сравнения коорди-*

нат" ранее в этой главе). Но только в данном случае мы хотим делать что-то другое с мячиком, по которому щелкнули, например ускорить или изменить направление его движения. (В версии этого примера, которую можно загрузить с сайта книги **http://www.prosetech/html5**, мячик, по которому щелкнули, изменяет движение.) Еще одну, довольно впечатляющую версию этого примера, можно исследовать по адресу **http://tinyurl.com/6byvnk5**. Здесь наведение курсора мыши на мячики разбрасывает их в разные стороны (как именно — зависит от того, каким образом наводится на них курсор), а если отвести курсор, мячики собираются в слово "Google". Если вам и этого мало, то другие примеры анимации можно посмотреть по адресу **http://www.blobsallad.se** (размножающиеся капли) и **http://tinyurl.com/crn3ed** (несколько банальные летающие звезды).

ЧАСТО ЗАДАВАЕМЫЙ ВОПРОС

Анимация на холсте для занятых (или ленивых)

Мне действительно нужно выполнять все вычисления самому?

Самый значительный недостаток анимации на холсте состоит в необходимости выполнять все вычисления самому разработчику. Например, если нужно, чтобы изображение двигалось от одной стороны холста к другой, разработчику надо рассчитать координаты каждого кадра, а потом нарисовать его в соответствующей позиции. А если требуется анимировать одновременно несколько предметов разными способами, объем и сложность необходимых для этого вычислений могут очень быстро выйти из под контроля. Поэтому намного легче жизнь разработчиков, использующих подключаемый модуль, такой как Flash или Silverlight. Обе технологии имеют встроенную систему анимации, которая позволяет разработчикам давать команды наподобие "переместить эту фигуру из этой точки в ту за 45 секунд" или, еще лучше, "переместить эту фигуру от верхнего края окна к нижнему, применяя эффект ускорения, вследствие которого фигура слегка отскакивает, достигнув нижнего края".

Нет сомнений, что однажды кто-то изобретет высокоуровневую систему анимации, которую можно будет наложить поверх холста. В идеальном случае такая система анимации позволит разработчику реализовывать желаемые эффекты, не требуя написания многочисленных страниц кода для математических вычислений. Вероятнее всего, что такой высокоуровневой системой анимации будет не усовершенствование основной спецификации HTML, а библиотека JavaScript. Но на данном этапе слишком рано говорить, какие средства разработки будут наиболее эффективными, с наименьшим количеством ошибок и с наивысшим уровнем поддержки.

Практический пример: игра "Лабиринт"

До сих пор мы изучали, каким образом применить к холсту некоторые ключевые методы программирования, чтобы демонстрировать интерактивные рисунки и анимацию. Эти методы позволяют использовать холст не только для простого рисования, но и создавать на его основе завершенные, автономные приложения, такие как игры или мини-приложения в стиле Flash.

На рис. 7.11 показан более амбициозный пример интерактивности и анимации, для реализации которого применяются все приобретенные нами на данный момент знания. Это простая игра, в которой пользователь должен провести значок через лабиринт. Значок начинает двигаться в определенном направлении после нажатия кла-

виши со стрелкой и продолжает перемещаться, пока не столкнется со "стеной" лабиринта. Для продолжения движения нужно нажать клавишу со стрелкой в направлении, в котором нет препятствий.

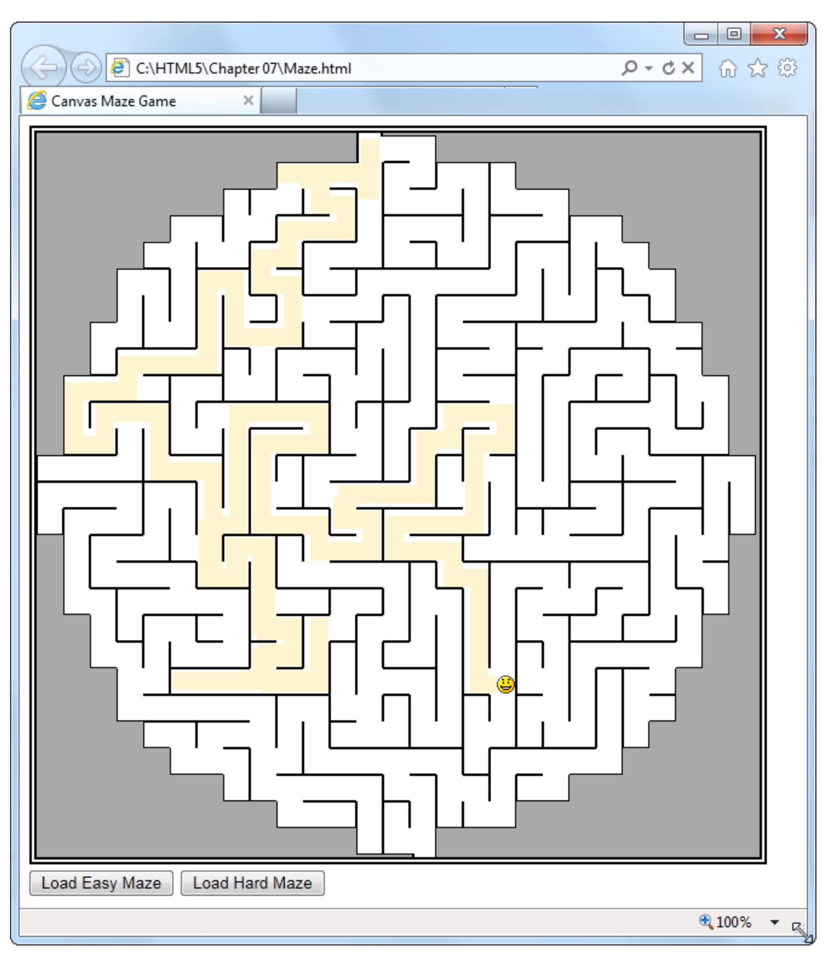


Рис. 7.11. Интерактивная игра в лабиринт на основе холста с анимацией. С точки зрения посетителя веб-страницы это забавная игра, а с точки зрения разработчика это эффективное использование возможностей холста HTML5 и искусного программирования на JavaScript

Но, как всегда, здесь есть свой подводный камень: использование холста для создания сложного приложения требует от разработчика перелопатить значительный объем кода. В следующих разделах мы изучим основные моменты создания этого приложения, но будьте готовы хорошенько попотеть с программированием JavaScript.

ПРИМЕЧАНИЕ

Этот пример можно запустить на локальном компьютере в Internet Explorer 9 или Opera 11.*x*. Но в других браузерах он будет работать только с веб-сервера. Во избежание этих проблем рекомендуется запускать пример с сайта книги — **http://www.prosetech.com/html5/**.

Подготовительные работы

Прежде чем анимировать что-либо, нам нужно создать для этого соответствующие условия, подготовив должным образом холст. Хотя весь лабиринт можно было бы нарисовать на холсте линиями и прямоугольниками, для этого потребовался бы внушительный объем кода, написание которого вручную будет чрезвычайно долгим и утомительным процессом. Для этого нужно составить логическую модель всего лабиринта, а потом вырисовывать каждую его часть отдельной операцией. Для этого подхода почти наверняка потребуется инструмент для автоматического создания кода рисования. Например, лабиринт можно будет нарисовать в Adobe Illustrator, а потом преобразовать его в код для холста с помощью какого-либо модуля расширения *(см. врезку "Часто задаваемый вопрос. Рисование на холсте для тех, кто ненавидит математику" главы 6)*.

Другой вариант — взять готовую графику лабиринта и прорисовать ее на холсте. Этот подход будет особенно легким, т. к. Интернет изобилует бесплатными страницами для создания лабиринтов. Найти такие страницы очень легко — просто выполните поиск в Google по словам "генератор лабиринтов" или "maze generator", и вы получите буквально тысячи ссылок. Выбрав понравившийся вам генератор, укажите несколько параметров (например, размер, форму, цвета, плотность и сложность лабиринта), нажмите кнопку **Создать**, и в считанные секунды вы получите рисунок лабиринта, который можно сохранить на своем компьютере.

В нашем примере используется готовый рисунок лабиринта. При загрузке страницы код отображает этот рисунок (хранящийся в файле maze.png) на холсте. Далее приводится код, который запускает этот процесс при загрузке страницы:

```
// Определяем глобальные переменные для холста и контекста рисования.
var canvas;
var context;

window.onload = function() {
  // Подготавливаем холст.
  canvas = document.getElementById("canvas");
  context = canvas.getContext("2d");

  // Рисуем фон лабиринта.
  drawMaze("maze.png", 268, 5);

  // При нажатии клавиши исполняем функцию processKey().
      window.onkeydown = processKey;
};
```

В действительности этот код не рисует фон лабиринта, а вызывает для этого другую функцию — `drawMaze()`.

Использование в этом примере отдельной функции рисования лабиринта означает, что он не ограничен одним видом лабиринта, а позволяет загрузить любую картин-

ку лабиринта. Для этого нужно просто вызвать функцию `drawMaze()` и передать ей название файла изображения лабиринта и координаты начала его прохождения. В следующем листинге приведен код этой функции:

```
// Отслеживаем текущую позицию значка.
var x = 0;
var y = 0;

function drawMaze(mazeFile, startingX, startingY) {
  // Загружаем изображение лабиринта.
  imgMaze = new Image();
  imgMaze.onload = function() {
    // Изменяем размер холста в соответствии
    // с размером изображения лабиринта
    canvas.width = imgMaze.width;
    canvas.height = imgMaze.height;

    // Рисуем лабиринт.
    var imgFace = document.getElementById("face");
    context.drawImage(imgMaze, 0, 0);

    // Рисуем значок.
    x = startingX;
    y = startingY;
    context.drawImage(imgFace, x, y);
    context.stroke();

    // Рисуем следующий кадр через 10 миллисекунд.
    setTimeout("drawFrame()", 10);
  };
  imgMaze.src = mazeFile;
}
```

В коде используется двухэтапный метод рисования изображения на холсте, рассмотренный в *разд. "Вставка в холст изображений" ранее в этой главе*. Сначала определяется функция для обработки события изображения `onLoad` и последующего отображения загруженного изображения на холсте. Потом устанавливается атрибут `img` объекта изображения, что загружает изображение и активирует код. Этот двухэтапный процесс немного посложнее, чем просто получение изображения из скрытого элемента `<img>` на странице, но он необходим для создания функции, позволяющей загружать любое изображение лабиринта.

После загрузки изображения лабиринта код подгоняет размер холста к размеру изображения, устанавливает значок в исходную позицию, а потом прорисовывает ее на холсте. Наконец, вызывается метод `setTimeout()`, чтобы начать показ кадров анимации.

ПРИМЕЧАНИЕ

Версия этого примера, которую можно загрузить на сайте книги (**http://www.prosetech.com/html5/**), немного утонченнее. Она разрешает пользователю загрузить новый лабиринт в любое время, даже в процессе его прохождения. Для этого в функцию `drawMaze()` добавлен дополнительный код, который останавливает значок (если в данный момент он находится в движении) и прекращает процесс анимации, прежде чем загрузить новый лабиринт и снова начать его прохождение.

Анимация значка

Процесс прохождения лабиринта начинается, когда пользователь нажмет одну из клавиш со стрелками. Например, при нажатии клавиши <↓> значок начинает двигаться вниз, пока не натолкнется на препятствие или не будет нажата другая клавиша.

Для этого в коде используются две глобальные переменные для отслеживания скорости значка, иными словами, количества пикселов, на которое она смещается по оси *x* или *y* в каждом кадре. Эти переменные называются `dx` и `dy`, точно так же, как в примере с падающими мячиками из *разд. "Анимация нескольких объектов" ранее в этой главе*, но с той разницей, что для их хранения не требуется массив, т. к. нужно отслеживать всего лишь один движущийся объект:

```
var dx = 0;
var dy = 0;
```

Когда пользователь нажимает какую-либо клавишу, холст вызывает функцию `processKey()`. Эта функция проверяет, не была ли нажата одна из клавиш со стрелкой, и если была, изменяет направление движения значка. Чтобы определить, какая именно клавиша со стрелкой была нажата, проверяется код нажатой клавиши. Например, код 38 соответствует клавише <↑>. Функция `processKey()` игнорирует все клавиши, за исключением клавиш со стрелками:

```
function processKey(e) {
  // Если значок находится в движении, останавливаем его.
  dx = 0;
  dy = 0;

  // Нажата стрелка вверх, начинаем двигаться вверх.
  if (e.keyCode == 38) {
      dy = -1;
  }

  // Нажата стрелка вниз, начинаем двигаться вниз.
  if (e.keyCode == 40) {
      dy = 1;
  }

  // Нажата стрелка влево, начинаем двигаться влево.
  if (e.keyCode == 37) {
```

```
      dx = -1;
  }

  // Нажата стрелка вправо, начинаем двигаться вправо.
  if (e.keyCode == 39) {
      dx = 1;
  }
}
```

Функция `processKey()` не меняет текущую позицию значка и не пытается обновить ее отображение на холсте. Эта задача осуществляется функцией `drawFrame()`, которая вызывается каждые 10 мс.

Код функции `drawFrame()` довольно простой, но всеохватывающий, решающий несколько задач. Прежде всего, он проверяет, движется ли значок в каком-либо направлении. Если не движется, то функция, по сути, ничего не делает:

```
function drawFrame() {
  if (dx != 0 || dy != 0) {
```

Если же значок движется, то функция `drawFace()` закрашивает желтым цветом текущую позицию значка, создавая, таким образом, след после продвижения. Потом значок перемещается в новую позицию:

```
context.beginPath();
context.fillStyle = "rgb(254,244,207)";
context.rect(x, y, 15, 15);
context.fill()

// Обновляем координаты значка.
x += dx;
y += dy;
```

Затем код вызывает функцию `checkForCollision()`, чтобы проверить новую позицию. (Код этой функции проверки на столкновение рассматривается в следующем разделе.) Если новая позиция не верна, это означает, что значок столкнулся с преградой, и ее нужно возвратить назад в старую позицию и прекратить движение:

```
if (checkForCollision()) {
  x -= dx;
  y -= dy;
  dx = 0;
  dy = 0;
}
```

Теперь все готово для рисования значка:

```
var imgFace = document.getElementById("face");
context.drawImage(imgFace, x, y);
```

Потом проверяется, не вышел ли значок за пределы лабиринта, т. е. прошел его. Если вышел, то выводится соответствующее сообщение:

```
  if (y > (canvas.height - 17)) {
    alert("You win!");
    return;
  }
}
```

В противном случае код устанавливает время ожидания для вызова метода `drawFrame()` опять 10 мс.

```
// Рисуем следующий кадр через 10 миллисекунд.
setTimeout("drawFrame()", 10); }
```

На данном этапе мы рассмотрели весь код для игры в лабиринт, за исключением логики функции `checkForCollision()`, которая выполняет проверку на столкновение значка с преградой. Эта тема обсуждается в следующем разделе.

Проверка на столкновение с использованием цвета пикселов

Ранее в этой главе вы увидели, как можно выполнять проверку на столкновение посредством математических вычислений. Но это можно сделать и другим способом. Вместо того чтобы просматривать коллекцию нарисованных объектов, можно исследовать цвет блока пикселов. В некоторых отношениях этот подход более простой, т. к. для него не требуется обрабатывать все объекты и код для отслеживания позиции объекта. Но он будет работать только в случае четко определенных предположений об исследуемых цветах.

ПРИМЕЧАНИЕ

Проверка на столкновение посредством тестирования пикселов является идеальным способом для применения в игре "Лабиринт". С помощью этого метода можно определить, когда значок столкнулся со стеной лабиринта черного цвета. Если бы не этот метод, то нужно было бы сохранить всю информацию о лабиринте в памяти, а потом определять, не перекрывают ли координаты значка линии стен лабиринта.

Метод проверки на столкновения посредством анализа цвета пикселов возможен благодаря предоставляемой холстом возможности манипулировать отдельными пикселами, из которых состоит любое изображение. Контекст рисования имеет три метода для манипулирования пикселами: `getImageData()`, `putImageData()` и `createImageData()`. Метод `getImageData()` применяется для захвата пикселов прямоугольной области холста. Захваченные пикселы можно изменить и вставить обратно в холст с помощью метода `putImageData()`. А метод `createImageData()` позволяет создать в памяти новый, пустой блок пикселов, которые можно изменить, а потом вставить в холст посредством метода `putImageData()`.

Чтобы лучше разобраться с этими методами для манипулирования пикселами, рассмотрим пример. Сначала с помощью метода `getImageData()` захватываются пикселы прямоугольной области холста размером 100×50 пикселов:

```
// Захватываем пикселы прямоугольника размером 100x50,
// левый верхний угол которого находится в точке (0, 0).
var imageData = context.getImageData(0, 0, 100, 50);
```

Потом посредством свойства `data` в числовой массив `pixels` помещаются данные о пикселах этого прямоугольника:

```
var pixels = imageData.data;
```

Если вы ожидаете, что каждый пиксел представляется одним числом, то ошибаетесь. Каждый пиксел представляется четырьмя числами — тремя для красной, зеленой и синей цветовых составляющих и одним для прозрачности. Поэтому, чтобы исследовать каждый пиксел, нам требуется цикл, который проходит через массив с шагом в четыре элемента:

```
// Перебираем все пикселы и инвертируем их цвет.
for (var i = 0, n = pixels.length; i < n; i += 4) {
  // Получаем данные для одного пиксела.
  var red =   pixels[i];
  var green = pixels[i+1];
  var blue =  pixels[i+2];
  var alpha = pixels[i+3];

  // Инвертируем цвета.
  pixels[i]   = 255 — red;
  pixels[i+1] = 255 — green;
  pixels[i+2] = 255 — blue;
}
```

Каждая составляющая цвета представляется числом в диапазоне от 0 до 255. В приведенном коде используется самый простой метод манипулирования пикселами — инверсия цвета. Результатом такой манипуляции является негативное изображение исходного рисунка.

Чтобы воочию увидеть результаты манипуляции, пикселы можно вставить в их исходное место в холсте (хотя с такой же легкостью их можно нарисовать в любом другом месте холста):

```
context.putImageData(imageData, 0, 0);
```

Методы манипулирования пикселами, определенно, обеспечивают разработчику высокий уровень контроля над содержимым. Но они также имеют и недостатки. Пиксельные операции медленные, в то время как объем пиксельных данных в среднем холсте огромен. Если захватить большую область изображения для исследования, то она будет содержать десятки тысяч пикселов. И если вы находите утомляющим рисование сложных фигур с помощью базовых составляющих, таких как прямые и кривые линии, то обработка отдельных пикселов — еще более утомительный процесс.

Но при всем этом методы манипулирования пикселами позволяют решать определенные задачи, решения которых иными способами было бы проблематичным. Например, они предоставляют самый легкий способ создания фрактальных узоров и фильтров для изображений в стиле Photoshop. А для нашей программы "Лабиринт" они дают возможность создать короткую процедуру для проверки столк-

новения значка со стеной лабиринта. Выполняющая эту работу функция `checkForCollision()` выглядит так:

```
function checkForCollision() {
  // Захватываем блок пикселов под значком, слегка
  // превышающий размер значка.
  var imgData = context.getImageData(x-1, y-1, 15+2, 15+2);
  var pixels = imgData.data;

  // Проверяем эти пикселы.
  for (var i = 0; n = pixels.length, i < n; i += 4) {
    var red =   pixels[i];
    var green = pixels[i+1];
    var blue =  pixels[i+2];
    var alpha = pixels[i+3];
    // Смотрим на наличие черного цвета стены, что указывает
    // на столкновение.
    if (red == 0 && green == 0 && blue == 0) {
        return true;
     }
     // Смотрим на наличие серого цвета краев, что указывает
     // на столкновение.
     if (red == 169 && green == 169 && blue == 169) {
         return true;
     }
  }
  // Столкновения не было.
  return false;
}
```

Итак, обсуждение программы "Лабиринт" завершено. Это были самый длинный код и самый объемный пример в этой книге. Вам, возможно, придется покопаться в нем немного (или освежить свои знания JavaScript), прежде чем вы все окончательно поймете в нем. Но приобретя такую уверенность, вы сможете применять подобные методы в собственных разработках на основе холста.

ПРАКТИЧЕСКИЕ ЗАНЯТИЯ ДЛЯ ОПЫТНЫХ ПОЛЬЗОВАТЕЛЕЙ
Сногсшибательные примеры работы с холстом

Диапазон творческих возможностей холста практически неограничен. Интернет содержит огромное множество еще более амбициозных примеров работы с холстом, которые демонстрируют высшее мастерство кодирования на HTML5. Далее приведен список некоторых веб-сайтов, демонстрирующих потрясающие примеры волшебства с холстом.

- **Образцы игр на основе холста.** Этот сайт содержит столько увлекательных примеров разработок на основе холста, что вы не сможете оторваться от экрана. Можно порекомендовать начать знакомство с этим сайтом с игры Mutant Zombie Masters или инструмента для построения графиков биржевых котировок TickerPlot. Адрес сайта — **www.canvasdemos.com**.

- **Карта знаний Википедии.** Это впечатляющее приложение на основе холста графически представляет статьи английской Википедии, где связанные темы соединяются тонкими линиями, похожими на паутину. При выборе новой темы соответствующая часть карты знаний помещается в центр страницы посредством плавной анимации. Адрес сайта — **http://en.inforapid.org**.

- **Трехмерный лабиринт.** В этой игре вы ходите с автоматом наперевес по простому трехмерному лабиринту, наподобие древней 3D-игры Wolfenstein, которая открыла повальное увлечение стрелялками в далеком 1992 г. Адрес сайта — **www.benjoffe.com/code/demos/canvascape**.

- **Шахматы.** В этой HTML5-игре вы можете испытать свои шахматные способности против компьютерного противника на нарисованной на холсте доске, которую можно представить как в трехмерном, так и в двухмерном измерении. Адрес сайта — **http://htmlchess.sourceforge.net/demo/example.html**.

8

ГЛАВА 8

Совершенствование стилей с помощью CSS3

Почти бессмысленно пытаться создать современный веб-сайт, не используя возможностей CSS (Cascading Style Sheet). Этот стандарт воткан в ткань Всемирной сети почти так же плотно, как и HTML. Каскадные таблицы стилей являются основным средством для любого типа деятельности в области веб-дизайна, будь то компоновка страниц, создание интерактивных кнопок и меню или простое декорирование. По сути, в то время как фокус HTML все больше смещается на содержимое и семантику *(см. разд. "Что такое семантические элементы?" главы 2)*, спецификация CSS стала центральной технологией *веб-дизайна*, т. е. оформления.

По пути, спецификация CSS стала более сложной и с более широким диапазоном охвата. Описание версии CSS 2.1 было в пять раз больше по сравнению с описанием предыдущей. К счастью, создатели стандарта CSS пошли иным путем, нежели обычное увеличение описания будущих возможностей. Они распределили усовершенствованные функции следующего поколения по наборам индивидуальных стандартов, называющихся *модулями*. Таким образом, разработчики браузеров могут реализовывать наиболее интересные и популярные части стандарта в первую очередь (что они и так уже делали, с модулями или без них). Всем этим новым модулям CSS было присвоено коллективное название CSS3 (обратите внимание на необычное отсутствие пробела, как и с HTML5).

Спецификация CSS3 содержит около 50 модулей разной степени завершенности. Возможности этих модулей широки: от предоставления декоративного оформления (такого как расширенные шрифты и анимация) до реализации более специализированного функционала (например, чтение текста или изменение стилей в зависимости от возможностей компьютера или мобильного устройства). В целом, эти модули содержат как надежно поддерживаемые возможности самыми последними версиями всех современных браузеров, так и экспериментальные.

В этой главе мы рассмотрим самые важные (и лучше всего поддерживаемые) части спецификации CSS3. Первым делом мы научимся выводить текст практически любым шрифтом. Потом мы узнаем, как адаптировать стили под браузерные окна разных размеров и разные типы устройств, такие как iPad и iPhone. Далее мы рассмотрим, как использовать тени, скруглять углы и улучшать внешний вид окон. Нако-

нец, мы научимся использовать переходы для создания утонченных эффектов при выделении элемента управления посредством наведения на него курсора, щелчком или с помощью клавиши <Tab>. (И все эти эффекты мы сможем сделать даже лучше с помощью еще двух возможностей CSS3 — трансформаций и прозрачности.)

Но прежде чем мы приступим к изучению и реализации этих захватывающих новых возможностей, нам нужно уделить время тому, как использовать эти возможности в разрабатываемых веб-сайтах, чтобы не потерять большую часть наших посетителей.

Современное использование CSS3

Спецификация CSS3 — это неоспоримое будущее в области декоративного оформления веб-страниц, и ее разработка еще далека от завершения. Большинство модулей все еще продолжает совершенствоваться и модифицироваться, и ни один браузер не поддерживает все модули. Это означает, что CSS3 испытывает такие же сложности, как и HTML5. Веб-разработчикам нужно решать, какие возможности использовать, а какие игнорировать, а также каким образом заполнить зияющие пробелы в браузерной поддержке.

Внедрять CSS3 в веб-сайт можно, по большому счету, используя три стратегии, которые рассматриваются в следующих разделах.

Примечание

Спецификация CSS3 не является частью спецификации HTML5. Эти два стандарта были разработаны отдельно друг от друга, разными людьми, работающими в разное время в различных местах. Но даже организация W3C призывает веб-разработчиков использовать HTML5 и CSS3 вместе, как часть одной новой волны современного веб-дизайна. Например, на странице этой организации для создания логотипов HTML5 (**www.w3.org/html/logo**) можно видеть, что W3C призывает разработчиков рекламировать CSS3 в ее логотипах HTML5.

Стратегия 1: используйте то, что можно

Логично использовать возможности с высоким уровнем поддержки на всех основных браузерах. В качестве примера одной из таких возможностей можно назвать применение веб-шрифтов *(см. разд. "Типография для Интернета" далее в этой главе)*. Используя шрифты правильного формата, эту функциональность можно заставить работать на таких старых браузерах, как Internet Explorer 6. К сожалению, очень немногие возможности CSS3 входят в эту категорию. Заворачивание текста работает практически на всех устройствах, а после доработки прозрачность можно заставить работать на старых браузерах. Но почти все иные возможности не будут работать на все еще популярных браузерах IE 7 и IE 8.

Примечание

Если не оговаривается иное, рассматриваемые в этой главе возможности работают на последних версиях всех основных браузеров, включая Internet Explorer 9. Но на более старых версиях IE они не работают.

Стратегия 2: рассматривайте возможности CSS3 как усовершенствования

У фанатов CSS3 есть боевой клич: "Веб-сайты не должны выглядеть абсолютно одинаково на всех браузерах". (У них даже есть свой веб-сайт — **http://DoWebsitesNeedToBeExperiencedExactlyTheSameInEveryBrowser.com**.) В этом они определенно правы.

Идея в основе этой стратегии заключается в использовании CSS3 для тонкой доработки страниц, причем эта доработка не повлияет на возможность просмотра основного содержимого и форматирования страницы в менее способных браузерах. Одним из примеров такой тонкой настройки является свойство `border-radius`, позволяющее скруглять углы рамок. Далее приводится пример указания этого свойства в правиле таблицы стилей:

```
header {
  background-color: #7695FE;
  border:           thin #336699 solid;
  padding:          10px;
  margin:           10px;
  text-align:       center;
  border-radius:    25px;
}
```

Браузеры, поддерживающие свойство `border-radius`, будут использовать его, а старые браузеры — просто игнорировать его, оставляя углы рамок квадратными (рис. 8.1).

Рис. 8.1. Internet Explorer 9 скругляет углы рамки верхнего колонтитула *(вверху)*, а Internet Explorer 8 игнорирует свойства `border-radius`, но применяет остальные свойства таблицы стилей *(внизу)*

Эта стратегия явно привлекательна, так она позволяет веб-дизайнерам манипулировать последними "игрушками" этой технологии. Но если слишком увлечься, она имеет и определенный недостаток. Несмотря на то, насколько хорошо веб-страница может выглядеть при просмотре в последней версии вашего любимого браузера,

запустив ее в одном из старых браузеров, которые используются значительной частью ваших посетителей, вы можете быть глубоко разочарованы ее намного менее впечатляющим внешним видом. А ведь вы хотите, чтобы ваш веб-сайт производил впечатление на всех, а не только на веб-фанатов, использующих лучшие браузеры.

По этой причине следует подходить к применению некоторых усовершенствований CSS3 с определенной долей осторожности. Ограничьтесь использованием возможностей, которые уже поддерживаются несколькими браузерами (и поддержка которых, по крайней мере, обещается в IE 10). И никогда не применяйте эти возможности так, чтобы производимое вашим сайтом впечатление резко менялось при его просмотре в старых браузерах.

Примечание

Что касается поддержки CSS3, Internet Explorer большой слабак в этой области. Существует воинствующее меньшинство веб-дизайнеров, которые считают, что веб-дизайнеры должны игнорировать этот браузер и применять возможности CSS3 как только они начинают поддерживаться другими браузерами. А как иначе оказывать давление на Microsoft и стимулировать усовершенствование Интернета? Такой подход вполне уместен, если основная цель вашего веб-сайта политическая, заключающаяся в продвижении передовых веб-стандартов. В противном случае следует иметь в виду, что сбрасывание со счета большого сегмента сетевой общественности плохо отразится на вас, т. к. человек все равно использует свой браузер (который вам может и не нравиться) для просмотра *вашей* работы.

Стратегия 3: добавляйте резервные решения с помощью Modernizr

Использование частично поддерживаемой возможности CSS3 — хорошая идея, если веб-сайт будет достойно выглядеть и без нее. Но иногда без этой возможности легко потерять важную часть дизайна своего веб-сайта, или же сайт может выглядеть просто неприглядно. Рассмотрим, например, что случится, если использовать многоцветную рамку, поддерживаемую только в браузере Firefox (рис. 8.2).

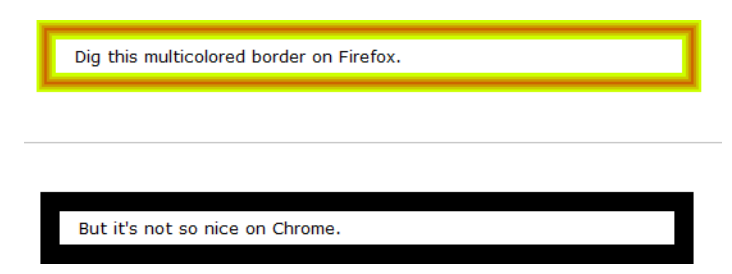


Рис. 8.2. Многоцветная рамка выглядит приятно в Firefox *(вверху)*. Но в Chrome эти настройки отображаются как толстая одноцветная рамка *(внизу)*, что не имеет никакой привлекательности

Иногда эту проблему можно решить, установив несколько свойств в правильном порядке. Здесь базовым методом будет установка сначала общих свойств, а за ними новых, которые замещают предыдущие свойства. Когда этот подход работает, он

удовлетворяет все браузеры — старые браузеры используют стандартные настройки, в то время как новые браузеры замещают эти настройки новыми. Далее показан пример применения этого метода для замены обычного фона градиентным:

```
.stylishBox {
  ...
  background: yellow;
  background: radial-gradient(ellipse, red, yellow);
}
```

Результаты применения этого правила показаны на рис. 8.3.

If you see a yellow background, you're lingering in the past.

If you see a radial gradient, you're rocking it, HTML5 style.

If you see a yellow background, you're lingering in the past.

If you see a radial gradient, you're rocking it, HTML5 style.

Рис. 8.3. *Вверху:* браузеры, которые не понимают CSS3, используют первую часть правила `stylishBox` и окрашивают фон сплошным желтым цветом. *Внизу:* браузеры, которые понимают CSS3, используют вторую часть правила и заполняют фон радиальным градиентом[1]

В некоторых случаях замещение свойств стиля не работает, т. к. устанавливаются *комбинированные* свойства. Примером комбинированного свойства является многоцветная рамка на рис. 8.2. Эффект многоцветности устанавливается свойством `border-colors`, но появляется только если установлена большая толщина рамки посредством свойства `border-thickness`. В браузерах, которые не поддерживают многоцветные рамки, толстая рамка одного цвета режет глаза независимо от цвета.

Одним из способов решения подобных проблем будет использование библиотеки JavaScript Modernizr, которая проверяет поддержку возможностей HTML5 конкретным браузером *(см. разд. "Определение возможностей с помощью Modernizr" главы 1)*. С помощью этой библиотеки можно определить альтернативные настройки

[1] По крайней мере, таков план. Этот пример работает не совсем так, как описано здесь, т. к. стандарт радиального градиента все еще находится в процессе пересмотра и исправлений. Чтобы получить описанный здесь результат, нужно использовать префиксы разработчиков браузеров, как описано в *разд. "Стили, специфичные для конкретных браузеров" далее в этой главе*.

стилей для браузеров, не поддерживающих свойства стилей, которые вы хотите использовать в первую очередь. Допустим, что нам нужно создать две версии рамки для верхнего колонтитула (см. рис. 8.1). Мы хотим использовать скругленные углы в браузерах, которые поддерживают эту возможность, и двойную рамку в браузерах, которые эту возможность не поддерживают. Добавив в разметку страницы ссылку на сценарий Modernizr, можно использовать следующую комбинацию правил таблицы стилей:

```
/* Настройки для всех браузеров, независимо от уровня поддержки CSS3 */
header {
  background-color: #7695FE;
  padding:    10px;
  margin:     10px;
  text-align: center;
}

/* Настройки для браузеров, которые поддерживают
   свойство border-radius. */
.borderradius header {
  border:        thin #336699 solid;
  border-radius: 25px;
}

/* Настройки для браузеров, которые не поддерживают
   свойство border-radius. */
.no-borderradius header {
  border: 5px #336699 double;
}
```

Этот набор правил работает следующим образом. В корневой элемент `<html>` страницы вставляется атрибут `class="no-js"`.

```
<html class="no-js">
```

Когда загружается Modernizr, этот сценарий выполняет быструю проверку на поддержку данным браузером ряда возможностей HTML5, JavaScript и CSS3. По результатам проверки сценарий вставляет в корневой элемент `<html>` страницы целую кучу классов, разделенный пробелами:

```
<html class="js flexbox canvas canvastext webgl no-touch geolocation
postmessage no-websqldatabase indexeddb hashchange history draganddrop
no-websockets rgba hsla multiplebgs backgroundsize borderimage borderradius
boxshadow textshadow opacity no-cssanimations csscolumns cssgradients
no-cssreflections csstransforms no-csstransforms3d csstransitions fontface
generatedcontent video audio localstorage sessionstorage webworkers
applicationcache svg inlinesvg smil svgclippaths">
```

Наличие в этом списке класса без префикса `no-` означает, что данный браузер поддерживает соответствующую возможность. Если же класс указан с префиксом `no-`, соответствующая возможность в данном браузере не поддерживается. Таким образом, в данном примере JavaScript поддерживается (`js`), но веб-сокеты не под-

держиваются (`no-websockets`). Из возможностей CSS3 поддерживается свойство `border-radius` (`borderradius`), но свойство `reflections` не поддерживается (`no-cssreflections`).

Эти классы можно вставить в селекторы таблицы стилей, чтобы отфильтровать настройки стилей в зависимости от предоставляемой поддержки. Например, если данный браузер поддерживает свойство `border-radius`, селектор `.borderradius header` получит все элементы `<header>` внутри корневого элемента `<html>`. В противном случае класс `.borderradius` будет отсутствовать, селектор ни с чем не совпадает, и правило не применится.

Но здесь есть своя загвоздка, заключающаяся в том, что Modernizr предоставляет классы только для определенного подмножества всех возможностей CSS3. В это подмножество входят некоторые наиболее популярные и развитые возможности CSS3, но немногоцветные рамки (см. рис. 8.2), т. к. эта возможность поддерживается только браузером Firefox. По этой причине лучше воздержаться от использования многоцветных рамок в своих разработках, по крайней мере на данном этапе.

ПРИМЕЧАНИЕ

С помощью Modernizr можно также создавать резервные решения на JavaScript. В данном случае нужно просто проверить значение соответствующего свойства объекта Modernizr, как при проверке на поддержку возможностей HTML5. Этот метод можно использовать, чтобы компенсировать отсутствие более продвинутых возможностей CSS3, таких как переходы или анимации. Но для этого требуется так много усилий и настолько разные модели, что для необходимых возможностей веб-сайта обычно лучше всего придерживаться решений только на основе JavaScript.

Стили, специфичные для конкретных браузеров

При разработке новых возможностей CSS часто приходится сталкиваться с дилеммой курицы и яйца. Для того чтобы усовершенствовать возможности, разработчикам нужно иметь отзывы и замечания об этих возможностях от разработчиков браузеров и веб-дизайнеров. Но чтобы разработчики браузеров и веб-дизайнеры могли предоставить такие отзывы и замечания, им нужно реализовать эти новые и несовершенные возможности. Это порождает цикл тестирований и предоставления отзывов по результатам тестирований, в результате которого многие усовершенствования принимают законченную форму. Но в процессе этого цикла синтаксис и реализация возможностей меняются. Такая ситуация порождает конкретную опасность применения какой-либо новой впечатляющей возможности неосведомленными веб-разработчиками, не осознающими, что будущие версии стандарта могут изменить правила, вследствие чего эта возможность больше не будет работать на веб-сайтах.

Чтобы предотвратить такое развитие событий, разработчики браузеров используют систему *префиксов разработчиков* (vendor prefixes), чтобы изменять названия свойств и функций CSS, пока они еще находятся в процессе разработки. Возьмем, например, новое свойство радиального градиента *(см. разд. "Градиенты" далее в этой главе)*. Чтобы использовать его в браузере Firefox, нужно установить разра-

боточную версию этого свойства, которое называется -moz-radial-gradient. Префикс разработчика -moz- (сокращение от Mozilla — организации, занимающейся проектом Firefox) обозначает свойство для браузера Firefox.

Для каждого браузера существует собственный префикс разработчика (табл. 8.1).

Таблица 8.1. *Префиксы разработчиков браузеров*

Префикс	**Браузер**
-moz-	Firefox
-webkit-	Chrome и Safari (в обоих браузерах используется один и тот же движок визуализации)
-ms-	Internet Explorer
-o-	Opera

Хотя префиксы разработчиков браузеров невероятно усложняют жизнь веб-разработчиков, для их использования есть хорошее основание. Разные разработчики браузеров добавляют поддержку возможностей в различное время, при этом часто используя разные предварительные версии одной и той же спецификации. Хотя все браузеры будут поддерживать одинаковый синтаксис для конечной версии, синтаксис свойств и функций, специфичных для конкретных разработчиков, часто бывает разным.

Поэтому, если вы хотите использовать в своем веб-сайте радиальный градиент уже сегодня, для того чтобы он правильно отображался во всех браузерах (включая грядущий Internet Explorer 10), вам нужно использовать раздутое правило CSS наподобие следующего:

```
.stylishBox {
  background: yellow;
  background-image: -moz-radial-gradient(circle, green, yellow);
  background-image: -webkit-radial-gradient(circle, green, yellow);
  background-image: -o-radial-gradient(circle, green, yellow);
  background-image: -ms-radial-gradient(circle, green, yellow);
}
```

В данном примере правило радиального градиента для каждого браузера построено с применением одинакового синтаксиса. Это указывает на то, что стандарт входит в стадию принятия окончательной формы, и разработчики браузеров вскоре смогут отбросить префиксы разработчиков и поддерживать свойство radial-gradient напрямую, как это уже делается сейчас для свойства corner-radius. Но синтаксис для данного свойства стал единообразным для разных браузеров сравнительно недавно, т. к. для старых версий Chrome использовался совершенно другой синтаксис.

ПРИМЕЧАНИЕ

Использование префиксов разработчиков браузеров — сложное и трудное занятие. Веб-разработчики разделились во мнении, являются ли эти префиксы необходимым злом для получения самых последних и лучших возможностей или же большим пре-

дупреждением для рассудительных разработчиков не связываться с ними. Но можно быть уверенным в одном: если не использовать префиксы разработчиков, значительная часть возможностей CSS3 в данное время будет недоступна.

Типография для Интернета

Среди всех новых захватывающих возможностей CSS3 трудно выбрать наилучшее. Но если бы пришлось выбирать только одну возможность, которая открывает дверь для лавины новых перспектив и которую можно использовать *прямо сейчас*, такой возможностью вполне могут быть веб-шрифты.

В прошлом веб-дизайнерам приходилось работать с ограниченным набором шрифтов, пригодных для веб-страниц. Под пригодными подразумеваются шрифты, о которых заведомо известно, что они работают на разных браузерах и в разных операционных системах. Но, как известно любому более-менее способному веб-дизайнеру, шрифты играют огромную роль в создании общего впечатления о документе. В зависимости от используемого шрифта, одно и то же содержимое может восприниматься как строго профессиональное, причудливое, старомодное или сверхсовременное.

Примечание

Разработчики браузеров не спешат реализовывать специальные веб-шрифты по объективным причинам. Прежде всего, это вопросы оптимизации, т. к. разрешение экранов намного хуже разрешения печатных документов. Если веб-шрифт не настроить должным образом для отображения на экране монитора, буквы маленького размера будут сливаться в одну расплывчатую линию. Кроме этого, в большинстве шрифты платные. Крупные компании, наподобие Microsoft, не горели желанием (что вполне понимаемо) добавлять функциональность, которая могла бы потакать загрузке веб-разработчиками установленных на компьютере шрифтов на веб-сайт без должного на то разрешения. Как мы увидим в следующем разделе, разработчики шрифтов нашли хорошие решения для каждой проблемы.

В CSS3 поддержка сложных шрифтов обеспечивается посредством возможности `@font-face`, которая применяется следующим образом:

1. Требуемый шрифт (или, более вероятно, несколько версий шрифта для поддержки разных браузеров) загружается на сайт.
2. Каждый шрифт регистрируется в таблице стилей с помощью команды `@font-face`.
3. Зарегистрированный шрифт используется в правилах стиля указанием его названия, точно так же, как и обычные веб-шрифты.
4. Когда браузер обнаруживает таблицу стилей, в которой используется специальный веб-шрифт, он загружает этот шрифт с сервера в свой кэш для временного хранения страниц и изображений. После этого браузер использует этот шрифт только для данной страницы или сайта (рис. 8.4). Если этот же шрифт указывается в другой странице, он должен быть зарегистрирован на этой странице и загружен на ее сервер, откуда он может быть загружен браузером в свой кэш.

В последующих разделах мы рассмотрим эти шаги более подробно.

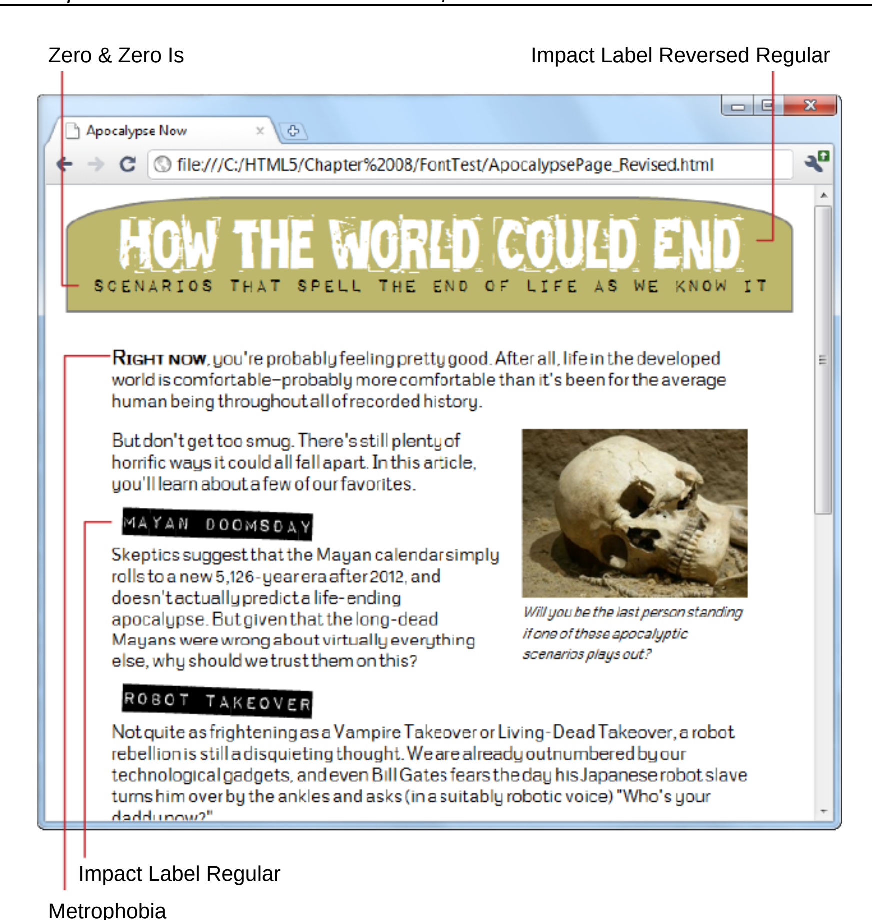


Рис. 8.4. В этой обновленной странице апокалипсиса используется набор из четырех разных шрифтов. Все эти шрифты бесплатные. Как загрузить их, объясняется в *разд. "Наборы шрифтов" далее в этой главе*

Примечание

Технически возможность `@font-face` не является новой. Она была частью спецификации CSS 2, но не вошла в CSS 2.1, когда разработчики браузеров не смогли договориться о сотрудничестве. Теперь в спецификации CSS3 предпринимаются новые усилия для того, чтобы сделать возможность `@font-face` универсальным стандартом.

Форматы веб-шрифтов

Хотя все современные браузеры поддерживают возможность `@font-face`, не все они поддерживают одинаковые *типы* файлов шрифтов. Например, Internet Explorer, который обеспечивает использование `@font-face` в течение многих лет, поддерживает только файлы типа EOT (Embedded OpenType, внедряемый OpenType). Этот формат предоставляет ряд преимуществ, например, в нем используется сжатие для уменьшения объема файла шрифтов, а также применяется строгое лицензирование для веб-сайтов, чтобы шрифт нельзя было украсть с одного сайта и использовать на

другом. Но формат EOT никогда ни пользовался большой популярностью и не используется никакими другими браузерами. Вместо него браузеры работают с более знакомыми стандартами шрифтов, применяемыми в компьютерных приложениях — TTF (TrueType) и OTF (OpenType PostScript). Кроме этого, существуют еще два типа отображения шрифтов — SVG и WOFF. В табл. 8.2 дано краткое описание всех этих форматов шрифтов.

Таблица 8.2. *Форматы внедряемых шрифтов*

Формат	**Описание**	**Используется в**
TTF (TrueType), OTF (OpenType PostScript)	Распространенные форматы шрифтов настольных компьютеров	Firefox (до версии 3.6), Chrome (до версии 6), Safari и Opera
EOT (Embedded OpenType)	Формат, специфичный для продуктов корпорации Microsoft. Не завоевал популярности у браузеров, за исключением Internet Explorer	Internet Explorer (до версии IE 9)
SVG (Scalable Vector Graphics)	Универсальный графический формат, который можно использовать для создания шрифтов. Дает хорошие, но не отличные результаты — медленно отображается и демонстрирует текст пониженного качества	Safari Mobile (на iPhone и iPad до iOS 4.2) и мобильные устройства под управлением операционной системы Android
WOFF (Web Open Font Format)	Возможно, единый формат шрифтов будущего. Поддерживается новыми версиями браузеров	Любой поддерживающий браузер, начиная с Internet Explorer 9, Firefox 3.6 и Chrome 6

В итоге можно сказать следующее: если вы хотите использовать возможность `@font-face` и поддерживать широкий диапазон браузеров, вам нужно предоставлять ваш шрифт в нескольких разных форматах. Как минимум, шрифт нужно предоставить в формате TTF или OTF (без разницы), EOT и SVG. Хорошо (но не обязательно) также предоставить шрифт в перспективном формате WOFF, может стать более популярным и лучше поддерживаемым в будущем. (Одним из достоинств этого формата является использование сжатых файлов, что сокращает время их загрузки.)

АВАРИЙНАЯ СИТУАЦИЯ
Устранение проблем

Даже если вы последуете всем приведенным ранее правилам и предоставите все требуемые форматы шрифтов, ожидайте некоторые загвоздки. В частности, с веб-шрифтами иногда возникают следующие проблемы.

- Многие шрифты выглядят плохо в все еще пользующейся популярностью операционной системе Windows XP, т. к. в настройках многих компьютеров с этой операционной системой отключено сглаживание (anti-aliasing). (А без применения сглаживания шрифты выглядят очень непривлекательно.)
- От пользователей поступали жалобы о проблемах с печатью определенных внедренных шрифтов из некоторых браузеров или в операционных системах.

* Некоторые браузеры страдают проблемой FOUT (Flash of Unstyled Text, вспышка нестилизованного текста). Это явление происходит, когда внедренный шрифт не успевает загрузиться вовремя, и страница отображается сначала в резервном шрифте, а потом воспроизводится повторно на встроенном шрифте. Эта проблема особенно заметна в старых версиях браузера Firefox. Если вас это сильно беспокоит, можете воспользоваться библиотекой JavaScript от Google, позволяющей разработчику определить резервные стили, которые используются вместо незагруженных внедренных шрифтов, таким образом, предоставляя ему полный контроль над воспроизведением текста в любое время. См. **http://code.google.com/apis/webfonts/docs/webfont_loader.html**.

Хотя эти небольшие проблемы иногда и возникают, большинство из них постепенно решается с выпуском новых версий браузеров. Например, браузер Firefox теперь сводит эффект FOUT к минимуму, ожидая до трех секунд, пока не загрузится внедренный шрифт, прежде чем использовать резервный шрифт.

Наборы шрифтов

Вы, наверное, уже задаетесь вопросом, где можно взять разные файлы шрифтов, которые могут вам потребоваться? Легче всего будет загрузить готовый набор шрифтов из Интернета, получив, таким образом, все требуемые файлы шрифтов. Недостаток этого подхода состоит в том, что ваш выбор ограничивается тем, что вы сможете найти в глобальной сети. Чтобы помочь вам в поиске наборов веб-шрифтов, можно порекомендовать воспользоваться одним из лучших сайтов шрифтов — Font Squirrel (**www.fontsquirrel.com/fontface**). На рис. 8.5 показано несколько шрифтов из предоставляемых на этом сайте наборов.

Набор шрифтов загрузится в виде файла, сжатого в формате ZIP, который содержит несколько файлов. Например, архив набора шрифтов Chantelli Antiqua, показанный на рис. 8.5, содержит следующие файлы:

```
Bernd Montag License.txt
Chantelli_Antiqua-webfont.eot
Chantelli_Antiqua-webfont.svg
Chantelli_Antiqua-webfont.ttf
Chantelli_Antiqua-webfont.woff
demo.html
stylesheet.css
```

Текстовый файл (Bernd Montag License.txt) содержит информацию об условиях лицензирования, которая, по большому счету говорит, что данный шрифт можно использовать бесплатно для любых целей, но нельзя продавать. Следующие четыре файла набора Chantelli Antiqua содержат разные форматы данного шрифта. (В зависимости от выбранного вами шрифта, архив может содержать дополнительные файлы для разных версий этого шрифта, например курсив, полужирный или жирный.) Наконец, файл demo.html содержит образец страницы с использованием этого шрифта, а файл stylesheet.css — правило таблицы стилей для применения шрифта в веб-странице.

Чтобы использовать шрифт Chantelli Antiqua в своей веб-странице, файлы разных форматов шрифта нужно загрузить на веб-сервер в папку этой веб-страницы. После этого шрифт нужно зарегистрировать, чтобы он был доступным для использования

в таблице стилей. Регистрация выполняется с помощью сложного правила `@font-face` в начале таблицы стилей, которое выглядит следующим образом (строки пронумерованы единственно с целью облегчения идентификации отдельных строк при рассмотрении их назначения):

```
1 @font-face {
2    font-family: 'ChantelliAntiquaRegular';
3    src: url('Chantelli_Antiqua-webfont.eot');
4    src: local('Chantelli Antiqua'),
5       url('Chantelli_Antiqua-webfont.woff') format('woff'),
6       url('Chantelli_Antiqua-webfont.ttf') format('truetype'),
7       url('Chantelli_Antiqua-webfont.svg') format('svg');
8 }
```

Строки приведенного кода выполняют следующие функции.

- **Строка 1:** выражение `@font-face` регистрирует шрифт для его дальнейшего применения в таблице стилей.

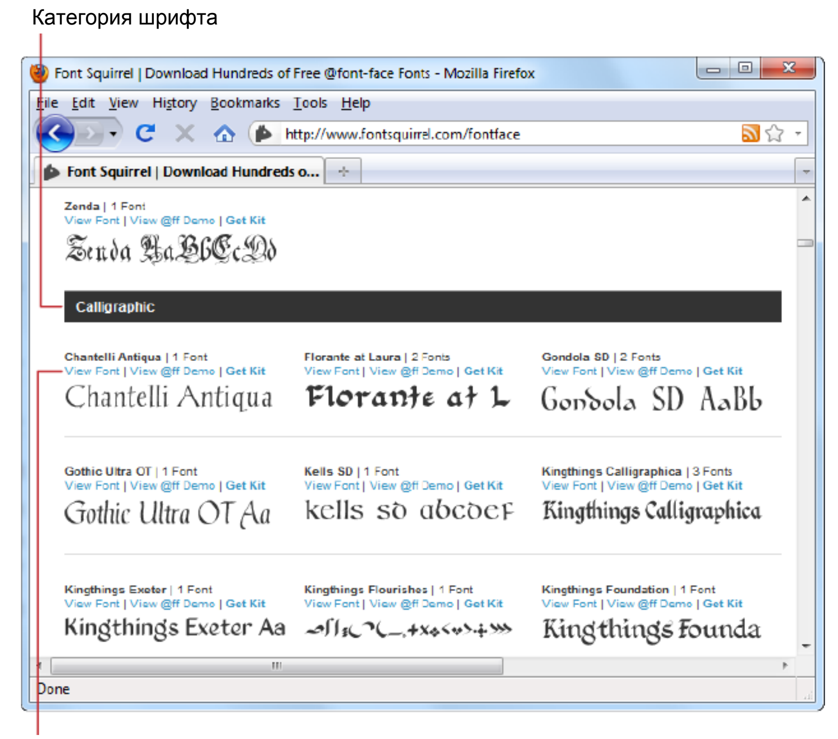


Рис. 8.5. Сайт Font Squirrel предоставляет для загрузки несколько сотен высококачественных шрифтов, организованных в разделы по категориям (такие как, например, **Calligraphic**, **Grunge** или **Retro**). Но самое приятное, что все эти шрифты бесплатные для любого использования, будь то на персональном компьютере для создания документов или на веб-странице в Интернете. Чтобы получше рассмотреть понравившийся шрифт, щелкните по ссылке **View Font**; чтобы увидеть, как выглядит текст в этом шрифте на веб-странице — по ссылке **View @ff Demo**; а чтобы загрузить его на свой компьютер — **Get Kit**

- **Строка 2:** шрифту можно присвоить любое название. Это название будет употреблено позже, при использовании шрифта.
- **Строка 3:** первым должен быть указан формат EOT, т. к. дальнейшая часть правила сбивает с толку Internet Explorer, и тот не обращает внимания на остальные форматы. Функция таблицы стилей `url()` указывает браузеру загрузить файл из обозначенного URL. Если шрифт размещен в одной папке с веб-страницей, то здесь можно просто указать название файла.
- **Строка 4:** функция `local()` указывает браузеру название шрифта, и если этот шрифт установлен на компьютере посетителя веб-странцы, браузер использует локальный шрифт. Но в редких случаях это может вызвать проблемы. Например, в зависимости от того, где установлен шрифт на компьютере посетителя, компьютеры Mac с OS X могут вывести предупреждение о нарушении безопасности, или же может загрузиться другой шрифт с таким же названием, как и ваш. По этой причине веб-разработчики иногда указывают явно несуществующее имя файла, чтобы браузер не нашел локального шрифта. В качестве простого имени такого типа можно использовать какой-либо бессмысленный символ наподобие `local('☺')`.
- **Строки 5—7:** последний шаг — это сообщить браузеру о других файлах шрифтов, которые он может использовать. Если имеется файл шрифта типа WOFF, укажите этот файл первым, т. к. данный формат предоставляет наилучшее качество шрифта. Следующим укажите файл шрифта формата TTF или OTF, а самым последним — файл формата SVG.

Совет

Конечно же, вводить правило `@font-face` от руки нет надобности, как и понимать все технические подробности, описанные выше. Можно просто скопировать это правило из файла stylesheet.css, который входит в состав архива веб-шрифтов.

Зарегистрировав веб-шрифт с помощью функции `@font-face`, вы можете использовать его в любой таблице стилей. Для этого используется уже знакомое нам свойство `font-family`, которому присваивается значение в виде названия семейства шрифтов, зарегистрированного с помощью функции `@font-face` (в строке 2). Далее приведен пример регистрации веб-шрифта (без подробностей) и последующего использования этого шрифта в правиле таблицы стилей:

```
@font-face {
  font-family: 'ChantelliAntiquaRegular';
  ...
}

body {
  font-family: 'ChantelliAntiquaRegular';
}
```

Это правило применяет веб-шрифт ко всей странице, хотя область его применения, конечно же, можно было бы ограничить определенными элементами или приме-

нить классы. Но шрифт нужно в обязательном порядке зарегистрировать *до того*, как использовать его в правиле таблицы стилей. Если выполнить эти шаги в обратном порядке, шрифт не будет работать должным образом.

Совет

Кроме пакетов шрифтов, веб-сайт Font Squirrel также предоставляет другие шрифты. Дополнительные шрифты можно найти в списке наиболее популярных шрифтов (открывается щелчком по ссылке **Popular**) или в списке недавно добавленных шрифтов (открывается щелчком по ссылке **Recent**). Эти списки содержат наборы веб-шрифтов, а также другие файлы бесплатных шрифтов, для которых нет вспомогательных файлов или которые нужно загрузить с другого веб-сайта. Если выбранный вами шрифт имеет только один формат, его можно преобразовать в другие форматы с помощью генератора шрифтов, предоставляемого сайтом Font Squirrel *(см. разд. "Использование своих шрифтов" далее в этой главе)*.

ЧАСТО ЗАДАВАЕМЫЙ ВОПРОС

Использование шрифта на локальном компьютере

Можно ли мне использовать один и тот же шрифт для веб-страниц и для распечатки документов?

Если вы нашли хороший шрифт для своего веб-сайта, вы, скорее всего, можете также использовать его на своем компьютере, например, чтобы создать какой-либо броский логотип. Современные компьютеры с Windows и Mac OS X поддерживают шрифты TrueType (ttf) и OpenType (otf). Все пакеты шрифтов на сайте Font Squirrel содержат шрифт в одном из этих форматов, обычно это формат TrueType. Чтобы установить этот шрифт в компьютере с Windows, извлеките его из архива, а затем щелкните на файле шрифта правой кнопкой мыши и в открывшемся контекстном меню выберите команду **Установить**. (Таким способом можно одновременно установить несколько выбранных файлов шрифтов.) На компьютере с Mac OS X запустите утилиту Font Book, а потом нажмите кнопку **Установить шрифт**.

Веб-шрифты Google

Еще один источник бесплатных веб-шрифтов — это служба Google Web Fonts. Ее особенность в том, что пользователю не нужно беспокоиться о форматах шрифтов, т. к. Google определяет браузер посетителя страницы и автоматически отправляет файл шрифта нужного формата.

Чтобы использовать шрифт Google в своих страницах, выполните такую последовательность шагов:

1. Откройте в браузере страницу **www.google.com/webfonts**. Нажмите кнопку **Start choosing fonts** (Начать выбор шрифтов). Откроется страница, содержащая длинный список имеющихся в наличии шрифтов (рис. 8.6).
2. Выберите способ просмотра шрифтов, перейдя на одну из вкладок вверху страницы — **Word** (Слово), **Sentence** (Предложение) или **Paragraph** (Абзац). Например, если вы ищете шрифт для заголовка, вам, скорее всего, следует выбрать вкладку **Word** или **Sentence**, чтобы посмотреть на шрифт в одном слове или предложении соответственно. Но если вам шрифт нужен для текста, то следует выбрать вкладку **Paragraph**, чтобы просмотреть, как этот шрифт будет выгля-

деть в большом фрагменте текста. При любой выбранной опции просмотра можно ввести свой текст образца и установить желаемый размер шрифта.

3. Настройте опции поиска. Если вы знаете название требуемого вам шрифта, введите его в поле поиска. В противном случае нужно будет просматривать все шрифты, прокручивая страницу вверх, что для латинских шрифтов может занять значительное время. Чтобы ускорить поиск, можно отсортировать и отфильтровать список согласно определенным критериям, например, наиболее популярные жирные шрифты без засечек. Расположение элементов управления для установки этих опций обозначены на рис. 8.6.

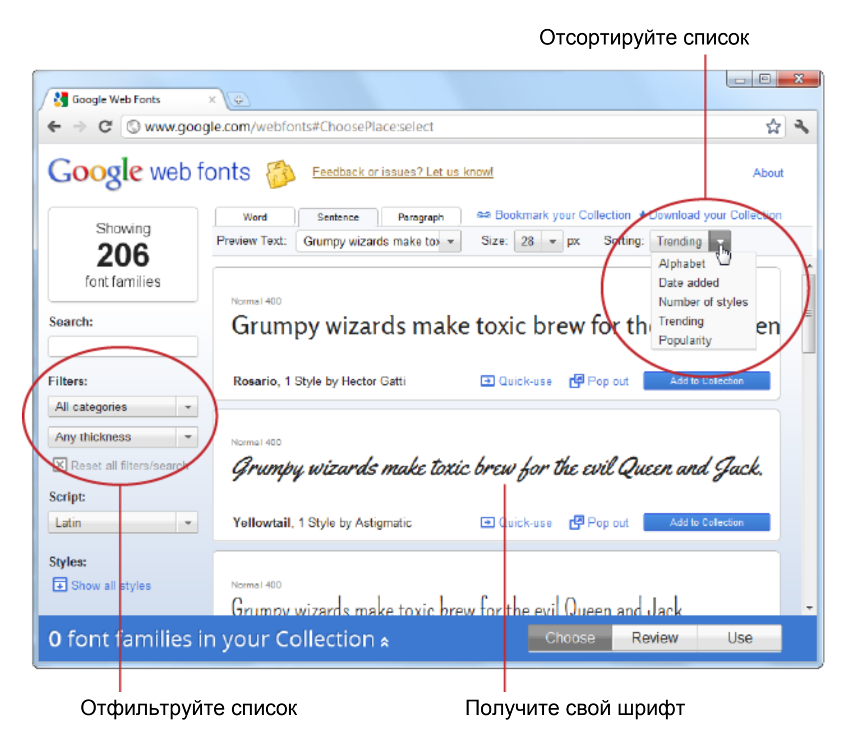


Рис. 8.6. Google обладает постоянно расширяющимся перечнем шрифтом. Процесс поиска подходящего шрифта можно облегчить, настроив опции сортировки и фильтрации списка шрифтов. Например, список можно отсортировать в алфавитном порядке или в порядке наибольшей популярности, а также наложить фильтр, чтобы список содержал только, например, шрифты с засечками, шрифты без засечек или курсивные шрифты

4. Обнаружив подходящий шрифт, щелкните по ссылке **Pop out**. Откроется новое окно, содержащее описание шрифта и все его подробности.

5. Если на этом этапе вы решите использовать данный шрифт, щелкните по ссылке **Quick-use** на исходной странице. Откроется окно, содержащее код, требуемый для использования данного шрифта. Код состоит из ссылки на таблицу стилей, которую нужно вставить в разметку вашей веб-страницы, и примера правила таблицы стилей, применяющего шрифт.

6. Вставьте в свою веб-страницу ссылку на таблицу стилей. Например, если выбран шрифт Metrophobic, в блок `<head>` вашей веб-страницы нужно вставить такую ссылку на таблицу стилей:

```
<link href="http://fonts.googleapis.com/css?family=Metrophobic"
      rel="stylesheet">
```

Данная таблица стилей регистрирует шрифт посредством функции `@font-face`, избавляя вас от необходимости выполнять эту работу самому. Но что лучше всего, Google предоставляет файлы шрифтов, что, опять же, избавляет вас от необходимости загружать их на свой веб-сайт.

Примечание

Следует помнить, что ссылку на таблицы стилей шрифтов Google нужно расположить первой в списке ссылок на таблицы стилей. Это позволит другим вашим таблицам стилей использовать шрифт Google.

7. Используйте шрифт, обращаясь к нему по его названию, в любое время. Например, далее приводится правило для применения должным образом зарегистрированного (см. шаг 6) шрифта Metrophobic в заголовках первого уровня, предоставляющее резервный шрифт на случай, если браузер не сможет загрузить основной шрифт:

```
hi {
  font-family: 'Metrophobic ', arial, serif;
}
```

ПРАКТИЧЕСКИЕ ЗАНЯТИЯ ДЛЯ ОПЫТНЫХ ПОЛЬЗОВАТЕЛЕЙ

Создание коллекции шрифтов

Изложенная ранее процедура указывает самый быстрый способ для создания разметки для требуемого кода. Но выбор шрифтов можно расширить, создав *коллекцию шрифтов*.

Коллекция шрифтов — это способ создания пакета шрифтов. Чтобы создать коллекцию шрифтов, нажмите кнопку **Add to Collection** справа от выбранного вами шрифта. По мере добавления шрифтов в коллекцию они отображаются в панели синего цвета в нижней части окна браузера.

Выбрав все необходимые шрифты, нажмите кнопку **Use** в этой панели. Откроется страница наподобие страницы **Quick-use** для одного выбранного шрифта, с той разницей, что она предоставляет информацию для создания ссылки на единую таблицу стилей, поддерживающую *все* шрифты из созданной вами коллекции шрифтов.

При создании коллекции шрифтов можно также использовать две ссылки, размещенные в правом верхнем углу страницы. Ссылка **Bookmark your Collection** позволяет скопировать ссылку на вашу коллекцию шрифтов, которую можно добавить в закладки браузера с тем, чтобы можно было возвратиться и редактировать эту коллекцию в будущем. А ссылка **Download your Collection** предоставляет возможность загрузить копии шрифтов на локальный компьютер, где их можно установить и использовать для создания и печати локальных документов.

Использование своих шрифтов

Некоторые веб-разработчики предъявляют высокие требования к своим шрифтам. Если вы принадлежите к числу таких разработчиков и у вас особые требования к шрифту для своих веб-страниц, то даже самая обширная библиотека бесплатных шрифтов может не содержать необходимого вам шрифта. К счастью, существует легкий способ адаптировать любой шрифт для использования в веб-страницах. С помощью правильного инструмента имеющийся у вас файл шрифта формата TTF или OTF можно преобразовать в другие форматы — EOT, SVG или WOFF.

Но прежде чем приступать к преобразованию, важно понимать следующее — обычные шрифты на вашем компьютере не являются бесплатными. Это означает, что без явного разрешения разработчика этого шрифта вы не можете просто взять шрифт со своего компьютера и использовать его на своем веб-сайте. Например, компании Microsoft и Apple покупают лицензии на определенные шрифты, которые они поставляют со своими операционными системами и приложениями, чтобы покупатели этих продуктов могли, скажем, создать квартальный отчет в текстовом редакторе. Но лицензия не позволяет покупателям этих продуктов использовать эти шрифты на веб-страницах и загружать их на веб-серверы.

Совет

Единственный способ узнать, нужно ли покупать лицензию на определенный шрифт для его использования в веб-странице — это связаться с его разработчиком. Некоторые разработчики шрифтов устанавливают стоимость своей лицензии на основе трафика, получаемого веб-сайтом. Другие могут позволить пользоваться своим шрифтом за номинальную плату или совсем бесплатно на определенных условиях (например, включения примечания мелким текстом о происхождении используемого шрифта или требования для некоммерческого сайта не предпринимать действий, направленных на зарабатывание больших сумм денег). Обращение к разработчику шрифта может также иметь положительный побочный эффект, т. к. умелые разработчики часто предоставляют версии своих шрифтов, оптимизированные для дисплеев.

Получив разрешение на использование определенного шрифта, его можно преобразовать в требуемый формат с помощью удобного инструмента, предоставляемого сайтом Font Squirrel (того самого, который предоставляет бесплатные наборы веб-шрифтов). Для преобразования шрифта откройте страницу **www.fontsquirrel.com/fontface/generator**. Откроется окно, показанное на рис. 8.7.

Первым делом нажмите кнопку **Add Fonts** (Добавить шрифты), чтобы загрузить файл шрифта со своего компьютера. Потом установите флажок **Yes, the fonts I'm uploading are legally eligible for web embedding** (Да, загружаемые мною шрифты отвечают юридическим требованиям для внедрения в веб-страницы, при условии, что эти шрифты отвечают лицензионным соглашениям), как описано в *разд. "Использование своих шрифтов" ранее в этой главе*. Наконец, нажмите кнопку **Download Your Kit**, чтобы сгенерировать набор веб-шрифтов и загрузить его на свой компьютер.

Этот набор веб-шрифтов точно такой же, как и наборы веб-шрифтов, предоставляемые для загрузки с этого веб-сайта. Он даже содержит таблицу стилей с разделом `@font-face`, а также тестовую веб-страницу.

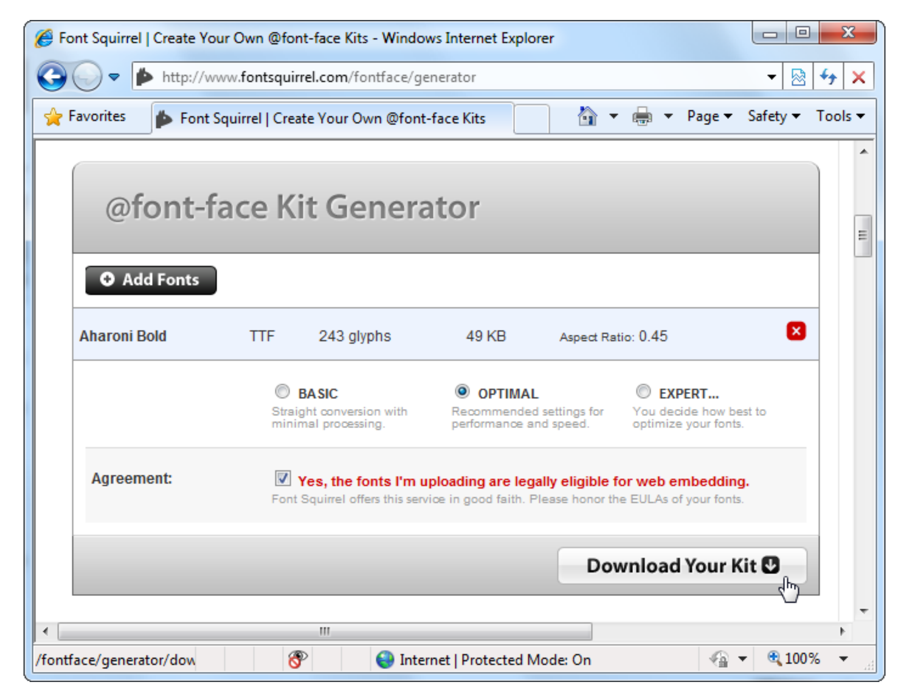


Рис. 8.7. Окно преобразователя шрифтов сайта Font Squirrel

Совет

Все еще не можете найти подходящий шрифт? Тогда посмотрите веб-сайт **http://webfonts.info**. На нем вы найдете ссылки на другие сайты, предоставляющие бесплатные шрифты, а также на профессиональные компании — разработчики шрифтов.

Размещение текста в несколько колонок

Инновации спецификации CSS3 в области отображения текста не ограничиваются новыми шрифтовыми возможностями. В нее также был добавлен абсолютно новый модуль для размещения текста в несколько колонок, таким образом предоставляя разработчикам гибкое средство для решения проблемы длинного текста без потери его читаемости. Создание нескольких колонок текста не требует почти никаких усилий и может быть выполнено двумя способами. Первый подход — это установить количество требуемых колонок с помощью свойства `column-count`, как показано в следующем коде:

```
article {
  text-align:   justify;
  column-count: 3;
}
```

На момент написания этой книги данный метод был совместим только с браузерами Opera. Для браузеров Firefox, Chrome и Safari это свойство используется с префиксом разработчика браузера:

```
article {
  text-align:            justify;
  -moz-column-count:     3;
  -webkit-column-count: 3;
  column-count:          3;
}
```

Как обычно, браузер Internet Explorer 9 не поддерживает эту возможность вообще, хотя, скорее всего, его следующая версия, IE 10, такую поддержку будет предоставлять.

Указание точного числа колонок лучше всего подходит для веб-страниц с компоновкой фиксированного размера. Но если размеры компоновки меняются в соответствии с размерами окна браузера, ширина колонок может оказаться слишком большой и текст в них будет трудно читать. Во избежание такой проблемы лучше не устанавливать точное число колонок, а дать указание браузеру, каким должен быть размер каждой колонки, используя свойство `column-width`, как показано далее:

```
article {
  text-align:            justify;
  -moz-column-width:     10em;
  -webkit-column-width: 10em;
  column-width:          10em;
}
```

Таким образом, браузер может создать такое количество колонок, которое необходимо для заполнения имеющегося пространства (рис. 8.8).

Примечание

Размер колонок можно указывать в пикселах, но лучше использовать для этого em-единицы[1], т. к. em-единицы адаптируются к текущему размеру шрифта. Таким образом, если посетитель веб-страницы увеличит размер шрифта в браузере, ширина колонки возрастет пропорционально размеру шрифта. В пикселах одна em-единица приблизительно вдвое больше размера шрифта. Например, для 12-пиксельного шрифта одна em-единица будет равна 24 пикселам.

Можно также настроить размер промежутка между колонками (свойство `column-gap`) и даже вставить вертикальную разделительную линию между ними (свойство `column-rule`). Дополнительную информацию обо всех возможностях форматирования колонок, включая способы управления разбиением текста между колонками и методы для перехода фигур и других элементов из одной колонки в другую, см. в полном стандарте по колонкам по адресу **www.w3.org/TR/css3-multicol**. К сожалению, на момент написания этих строк эти продвинутые возможности не поддерживались ни одним браузером.

[1] Буква `m` как мера напечатанного на строке, странице (прежде за единицу измерения принималась площадь, занимаемая этой буквой).

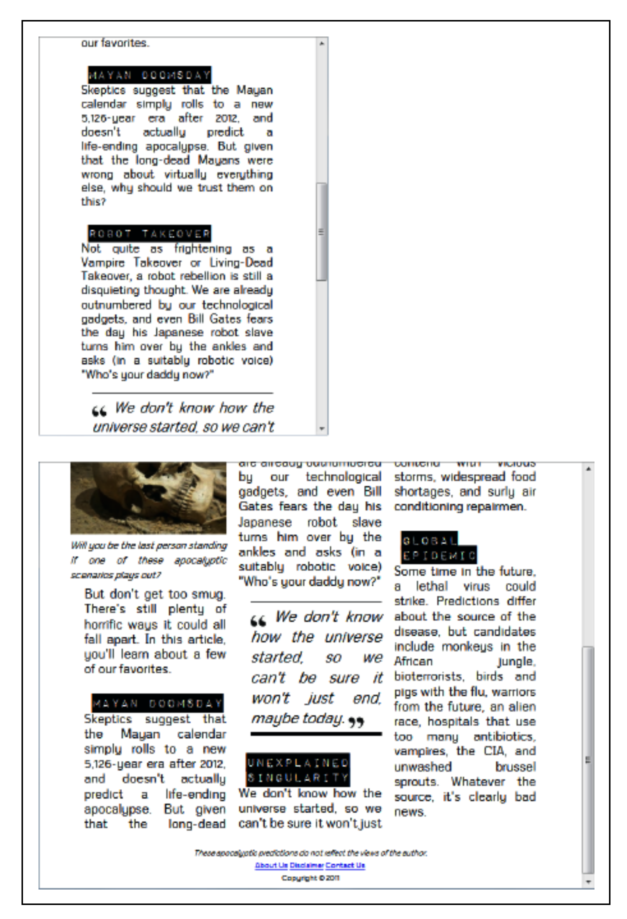


Рис. 8.8. В узком окне браузера можно поместить только одну колонку текста *(вверху)*, но в более широком окне количество колонок увеличивается *(внизу)*

Адаптация к разным устройствам

Если вы когда-либо работали в Интернете более-менее продолжительное время с мобильного устройства, то, несомненно, обнаружили, что крохотный экранчик (каким бы большим он не был для мобильного устройства) — не лучшее окно в Интернет. Конечно же, практически любую веб-страницу можно просмотреть, прокручивая ее во всех направлениях и масштабируя ее туда и обратно. Но вряд ли такой процесс будет способствовать получению положительного восприятия этой страницы. Ситуация намного улучшается, если вы заходите на сайт, разработанный специально для мобильных устройств, размер содержимого которого масштабируется под размеры экрана вашего устройства.

В настоящее время создание разработчиком специальных версий одного и того же веб-сайта для конкретных устройств, таких как iPhone или iPad, не будет чем-то необычным. Эти сайты, как правило, размещаются на других доменах, чем обычная версия сайта. Например, домен обычного сайта газеты "New York Times" —

http://www.nytimes.com, а мобильного сайта — **http://rn.nytimes.com**. Но с этим подходом есть своя закавыка: по мере того, как выход в Интернет с мобильных устройств становится все более популярным, а мобильные устройства становятся все более разнообразными, веб-разработчики могут оказаться в ситуации, где сопровождение всех этих сайтов под конкретные типы мобильных устройств может потребовать слишком больших усилий или вообще не под силу.

Опять же, создание отдельных версий сайта для каждого типа мобильного устройства не является единственным способом решения проблемы разных мобильных устройств. Другой подход — разработать код веб-сервера, который исследует каждый запрос, вычисляет, какой браузер подал этот запрос, и отправляет содержимое соответствующего формата. Такое решение, безусловно, великолепно, но только если у вас есть время и необходимые навыки. Но было бы замечательно иметь простой механизм, который бы подстраивал стили вашей веб-страницы под разные типы устройств, не требуя для этого ни инфраструктуры веб-приложения, ни серверного кода.

Представляем *запросы о возможностях отображения* (media queries). Эта возможность CSS3 предоставляет простой способ изменять стили веб-страницы для разных устройств и разных настроек просмотра. Аккуратное использование таких запросов может помочь вам предоставить свою веб-страницу любому устройству — от сверхширокого экрана домашнего компьютера до экрана смартфона iPhone. И все это без изменения ни единой строчки кода HTML.

Запросы о возможностях отображения

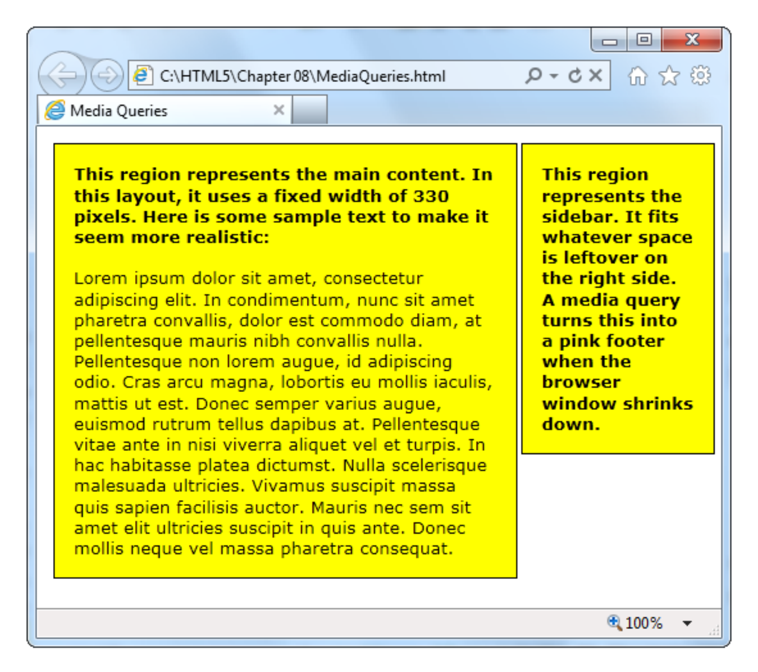
Принцип работы запросов о возможностях основан на получении ключевой информации об устройстве, на котором просматривается веб-страница, такой как, например, размер экрана, его разрешающая способность, возможности цветовоспроизведения и т. п. На основе этих сведений можно применять разные стили форматирования или даже подключать совершенно другие таблицы стилей. Результаты работы запросов о возможностях показаны на рис. 8.9.

УГОЛОК НОСТАЛЬГИИ
Типы носителей CSS

Довольно интересно, создатели CSS пытались решить проблему разных типов устройств воспроизведения в CSS 2.2 с помощью *типов носителей* (media types). Вы, возможно, уже пользуетесь этим стандартом, чтобы предоставлять разные таблицы стилей для вывода содержимого на разные устройства, как показано в следующем коде:

```
<head>
  ...
  <!-- Используем эту таблицу стилей для отображения
       страницы на экране. -->
  <link rel="stylesheet" media="screen" href="styles.css">

  <!-- Используем эту таблицу стилей для распечатки страницы. -->
  <link rel="stylesheet" media="print" href="print_styles.css">
</head>
```



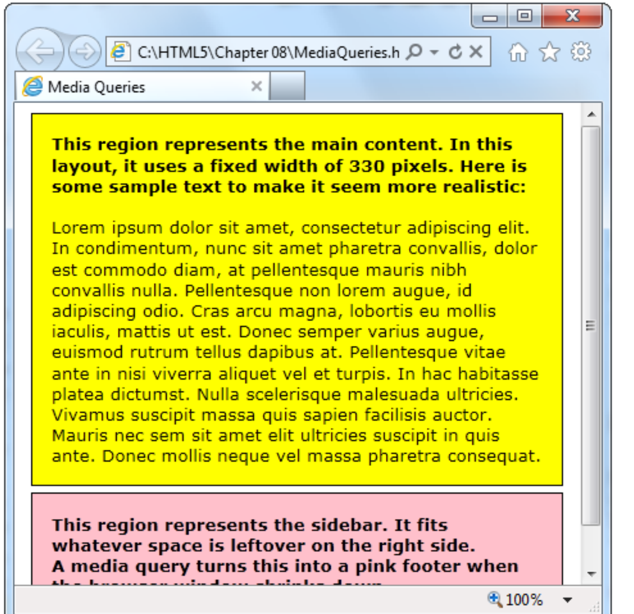


Рис. 8.9. Одна и та же страница, просматриваемая в широком и узком окне (*вверху* и *внизу* соответственно). Когда размер окна уменьшается, механизм запросов о возможностях автоматически заменяет часть правил таблицы стилей для страницы, превращая боковую панель в нижний колонтитул. Страницу даже не нужно обновлять

> Атрибут `media` также принимает значение `handheld`, предназначенное для мобильных устройств с экраном небольшого размера и низкоскоростным подключением. Большинство мобильных устройств пытаются обращать внимание на атрибут `media` и использовать таблицу стилей `handheld`, если такая имеется. Но этот подход изобилует разными огрехами, и возможности атрибута `media` весьма ограничены для работы с широким диапазоном устройств, подключенных к Интернету в настоящее время.

При использовании запросов о возможностях первое, что необходимо сделать, — это выбрать свойство, которое нужно исследовать. На рис. 8.9 ключевой информацией является свойство `max-width`, которое получает текущий размер страницы в окне браузера. Еще более полезным будет свойство `max-device-width`, которое проверяет максимальную ширину экрана. Если это значение небольшое, очевидно, что мы имеем дело со смартфоном или другим подобным устройством небольшого размера.

Чтобы использовать запросы о возможностях, легче всего начать со стандартной версии веб-сайта, а потом замещать определенные части разметки. В примере на рис. 8.9 содержимое страницы разбито на два блока:

```
<article>
  ...
</article>
<aside>
  ...
</aside>
```

А таблица стилей начинается с двух правил, по одному для каждого блока:

```
article {
  border:     solid 1px black;
  padding:    15px;
  margin:     5px;
  background: yellow;
  float:      left;
  width:      330px;
}

aside {
  border:      solid 1px black;
  padding:     15px;
  margin:      5px;
  background:  yellow;
  position:    absolute;
  float:       left;
  margin-left: 370px;
}
```

Эти правила реализуют стандартную двухстолбцовую компоновку страницы, где колонка фиксированной ширины в 330 пикселов располагается слева, а плавающая боковая панель занимает все оставшееся пространство справа. (Конечно же, в своих

примерах вы можете использовать любые компоновки, управляемые с помощью CSS.)

Фокус заключается в определении отдельной части таблицы стилей, которая активируется для определенного значения запроса о возможностях. Синтаксис этого определения следующий:

```
@media (имя-свойства-запроса-о-возможностях: value) {
  /* Новые стили вставляются сюда. */
}
```

В данном примере этот новый набор стилей активируется, когда ширина окна браузера становит 480 пикселов или меньше. Это означает, что в нашей таблице стилей нам требуется раздел, который выглядит так:

```
@media (max-width: 480px) {
  ...
}
```

Совет

В настоящее время самыми популярными свойствами запросов о возможностях являются `max-device-width` (для создания мобильных версий веб-страниц), `max-width` (для применения разных стилей в зависимости от текущего размера окна браузера) и `orientation` (для изменения компоновки страницы, в зависимости от расположения устройства iPad — горизонтального или вертикального). Но спецификация запросов о возможностях определяет еще несколько других свойств, полный список которых см. на **www.w3.org/TR/css3-mediaqueries**.

Хотя внутри блока запроса о возможностях можно вставить все, что угодно, в этом примере просто добавляются новые правила таблицы стилей для элементов `<article>` и `<aside>`:

```
@media (max-width: 480px) {
  article {
    float: none;
    width: auto;
  }

  aside {
    position:    static;
    float:       none;
    background:  pink;
    margin-left: 5px;
  }
}
```

Эти стили применяются в дополнение к уже определенным обычным стилям. Поэтому может потребоваться сбросить уже измененные свойства в их значения по умолчанию. В этом примере стили запроса о возможностях присваивают свойству `position` значение `static`, свойству `float` — значение `none`, а свойству `width` — зна-

чение `auto`. Это значения по умолчанию, которые были изменены исходным правилом стиля для боковой панели.

ПРИМЕЧАНИЕ

Браузеры, которые не понимают запросов о возможностях, такие как Internet Explorer, просто игнорируют новые стили и применяют исходные стили независимо от размера окна браузера.

При желании можно добавить другой раздел запроса о возможностях, который замещает эти правила еще меньшим размером. Например, следующий блок активирует применение новых правил, когда окно браузера сужается до 250 пикселов:

```
@media (max-width: 250px) {
  ...
}
```

Но не забывайте, что эти правила замещают все, что было применено до этого, иными словами, совокупный набор свойств, установленный обычными стилями и блоком стилей запросов о возможностях для ширины окна менее 450 пикселов. Если это кажется слишком заумным, не впадайте в отчаяние — мы научимся обходить это с помощью более точно определенных запросов о возможностях в следующем разделе.

СОВЕТ

Для определения мобильных устройств наподобие смартфонов нужно использовать свойство `max-device-width`, а не свойство `max-width`, т. к. свойство `max-width` использует размер окна просмотра (viewport) телефона, т. е. блока веб-страницы, который пользователь веб-фона может просматривать с помощью прокруток. Типичное окно просмотра вдвое шире собственно экрана устройства. Более подробное объяснение этого вопроса (и рисунки, иллюстрирующие принцип работы окна просмотра) см. в статье "Quirks mode" по адресу **http://tinyurl.com/yyec93n**. А запросы о возможностях для мобильных устройств рассматриваются более подробно в *разд. "Распознавание мобильных устройств" далее в этой главе*.

Продвинутые запросы о возможностях

Иногда желательно применить даже более специфичные стили, зависящие от нескольких условий, как показано в этом примере:

```
@media (min-width: 400px) and (max-width: 700px) {
  /* Эти стили применяются к окнам шириной от 400 до 700 пикселов */
}
```

Это особенно удобно, когда нужно применить несколько наборов взаимно исключающих стилей, но нет желания мучиться с несколькими слоями перекрывающихся правил. Вот пример применения такого набора стилей:

```
/* Здесь идут обычные стили. */

@media (min-width: 600px) and (max-width: 700px) {
  /* Замещаем стили для окон шириной 600-700 пикселов. */
}
```

```
@media (min-width: 400px) and (max-width: 599.99px) {
  /* Замещаем стили для окон шириной 400-600 пикселов. */
}

@media (max-width: 399.99px) {
  /* Замещаем стили для окон шириной менее 400 пикселов. */
}
```

Теперь для окна браузера шириной 380 пикселов будут применяться два набора стилей: стандартные стили и стили в последнем блоке `@media`. Ответ на вопрос, упрощает или усложняет этот подход работу разработчика, зависит от того, чего именно он желает добиться. Если вы используете сложные таблицы стилей и часто меняете их, показанный здесь подход неперекрывающихся стилей часто будет самым простым способом.

Обратите внимание, что необходимо быть осторожным, чтобы правила неожиданно не перекрывали друг друга. Например, если в одном правиле установить максимальную ширину в 400 пикселов, а в другом правиле указать те же 400 пикселов, но для минимальной ширины, то в одной точке обе настройки стилей будут совмещены. Слегка неуклюжим решением этой проблемы будет использование дробных значений, как значение `399.99` пикселов в примере выше.

Можно еще использовать ключевое слово `not`. Функционально, по сути, это такая же таблица стилей, но, может быть, более понятная:

```
/* Здесь идут обычные стили. */

@media (not max-width: 600px) and (max-width: 700px) {
  /* Замещаем стили для окон шириной 600-700 пикселов. */
}

@media (not max-width: 400px) and (max-width: 600px) {
  /* Замещаем стили для окон шириной 400-600 пикселов. */
}

@media (max-width: 400px) {
  /* Замещаем стили для окон шириной менее 400 пикселов. */
}
```

В этих примерах есть еще один уровень замещения стилей, который нужно иметь в виду, т. к. каждый раздел `@media` начинается не с правил стилей для запросов о возможностях, а с обычных правил таблиц стилей. В зависимости от ситуации, может быть предпочтительней полностью разделить логику стилей (например, чтобы мобильное устройство получило свой, полностью независимый набор стилей). Для этого нужно использовать запросы о возможностях с внешними таблицами стилей, как рассматривается далее.

Полная замена таблицы стилей

Блок `@media` удобен для небольших корректировок, т. к. он позволяет содержать все стили вместе в одном файле. Но для более значительных изменений может быть легче просто создать полностью отдельную таблицу стилей, а потом с помощью запроса о возможностях создать ссылку на нее:

```
<head>
  <link rel="stylesheet" href="standard_styles">
  <link rel="stylesheet" media="(max-width: 480px)"
        href="small_styles.css">
</head>
```

Хотя браузер загрузит со страницей и вторую таблицу стилей (small_styles.css), она не будет применена, если только ширина окна браузера не отвечает указанному минимуму.

Как и в предыдущем примере, новые стили будут замещать уже установленные стили. В некоторых случаях уместно использовать полностью отдельные, независимые таблицы стилей. Сначала к обычной таблице стилей нужно добавить запрос о возможностях, чтобы она применялась только для окон с большой шириной:

```
<link rel="stylesheet" media="(min-width: 480.01px)"
      href="standard_styles">
<link rel="stylesheet" media="(max-width: 480px)"
      href="small_styles.css">
```

Проблема с этим подходом состоит в том, что браузеры, которые не понимают запросов о возможностях, будут игнорировать обе таблицы стилей. Эту проблему можно решить для старых версий Internet Explorer, опять добавив основную таблицу стилей, но в условных комментариях:

```
<link rel="stylesheet" media="(min-width: 480.01px)"
      href="standard_styles">
<link rel="stylesheet" media="(max-width: 480px)"
      href="small_styles.css">
<!--[if lt IE 9]>
  <link rel="stylesheet" href="standard_styles">
<![endif]-->
```

Но все равно остается один небольшой пробел — версии Firefox более ранние, чем 3.5, не понимают запросов о возможностях и не используют раздел условных комментариев. Этот пробел можно было бы заполнить, определяя браузер в коде страницы, а потом подменяя новую страницу посредством кода JavaScript, но это будет очень громоздкое решение. К счастью, старые версии Firefox встречаются все реже и реже.

Кстати, запросы о возможностях отображения можно объединять с запросами о типах носителей, рассмотренных во *врезке "Уголок ностальгии. Типы носителей CSS" ранее в этой главе*. В таких случаях запрос о типе носителя всегда нужно ставить первым и не брать его в скобки. Например, таблицу стилей только для печати страницы определенной ширины можно создать следующим образом:

```
<link rel="stylesheet" media="print and (min-width:  25cm)"
      href="NormalPrintStyles.css">
<link rel="stylesheet" media="print and (not min-width: 25cm)"
      href="NarrowPrintStyles.css">
```

Распознавание мобильных устройств

Как мы уже узнали, выяснить, где просматривается веб-страница — на экране обычного компьютера или мобильного устройства, можно с помощью запроса о возможностях отображения, содержащего свойство `max-device-width`. Но какие значения ширины следует использовать для этого свойства?

Если вы пытаетесь идентифицировать смартфоны, проверяйте свойство `max-device-width` на значение, равное 480 пикселам. Это наилучшее, общеприменимое правило, которое определяет существующие сегодня телефоны iPhone и Android:

```
<link rel="stylesheet" media="(max-device-width: 480px)"
      href="mobile_styles.css">
```

Но если вы фанат железа, это правило может вызвать у вас подозрение, т. к. современные мобильные устройства оснащены крохотными экранами со сверхвысокой разрешающей способностью. Например, размер экрана iPhone 4 — 960×640 пикселов. В связи с этим можно подумать, что для этих устройств нужно использовать большее значение ширины экрана. Но, как это ни удивительно, это не так. Большинство смартфонов продолжает сообщать, что ширина их экрана — 480 пикселов, даже если они в действительности оснащены экраном высокого разрешения. Это объясняется применением этими устройствами поправочного коэффициента, который называется *соотношением пикселов* (pixel ratio). Например, в iPhone 4 каждому пикселу CSS соответствует в ширину два физических пиксела, т. е. соотношение пикселов равно 2. В действительности можно создать запрос о возможностях, который определяет iPhone 4, но игнорирует более старые версии iPhone:

```
<link rel="stylesheet"
      media="(max-device-width: 480px) and
             (-webkit-min-device-pixel-ratio: 2)"
      href="iphone4.css">
```

А iPad представляет особую проблему: пользователи могут поворачивать его, сориентировав экран горизонтально или вертикально. Но хотя это меняет значение ширины окна `max-width`, значение ширины экрана `max-device-width` не меняется. Как и книжной, так и альбомной ориентации экрана iPad сообщает, что его ширина экрана равна 768 пикселов. К счастью, если требуется менять стили в зависимости от ориентации экран iPad, в запросах о возможностях свойство `max-device-width` можно применять совместно со свойством `orientation`:

```
<link rel="stylesheet"
   media="(max-device-width: 768px) and (orientation: portrait)"
   href="iPad_portrait.ess">
<link rel="stylesheet"
   media="(max-device-width: 768px) and (orientation: landscape)"
   href="iPad_landscape.css">
```

Конечно же, это правило можно применять не только с устройствами iPad, но и с другими устройствами с подобными размерами экрана (в данном случае 768 пикселов или меньше).

ПРИМЕЧАНИЕ

Сами по себе запросы о возможностях отображения недостаточны, чтобы сделать обычный веб-сайт дружелюбным к мобильным устройствам. Нужно еще позаботиться о пропускной способности и восприятии пользователя. В отношении пропускной способности разумно использовать изображения меньшего размера, занимающие меньший объем. Это можно сделать, снабдив элементы фоновыми изображениями и установив эти изображения в таблицах стилей. Но этот подход очень сложен для веб-сайтов, на которых используется большое число изображений. Для лучшего восприятия контента пользователем нужно рассмотреть вариант разбиения содержимого на блоки меньшего размера (чтобы требовалось меньше прокруток) и по возможности отказа от эффектов и интерактивных операций, которые трудно осуществлять через сенсорный интерфейс (например, всплывающие меню).

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ

Запросы о возможностях воспроизведения видео

Веб-сайты для мобильных устройств явно отличаются от веб-сайтов для настольных компьютеров способом использования видеосодержимого. Мобильный веб-сайт может содержать видео, но, как правило, будет использовать видеоокно и видеофайл меньшего размера. Причины здесь очевидны — мобильные браузеры используют не только более медленные и более дорогие подключения для загрузки видео, они также обладают менее мощным аппаратным обеспечением для его воспроизведения.

Применяя только что рассмотренную технологию запросов о возможностях, размер элемента <video> можно с легкостью изменить, чтобы видео можно было посмотреть на мобильном устройстве. Но второй важный шаг — создание ссылки на видеофайл меньшего размера — представляет более сложную задачу.

HTML5 предлагает решение этой задачи путем добавления атрибута media непосредственно в элемент <source>. Как мы узнали в *главе 5*, элемент <source> указывает файл мультимедиа для воспроизведения элементом <video>. Добавление атрибута media позволяет ограничить воспроизведение определенных файлов мультимедиа на конкретных типах устройств.

Далее приведен пример кода, позволяющий проиграть файл butterfly_mobile.mp4 на устройствах с небольшим экраном. Для других устройств предоставляется файл butterfly.mp4 или butterfly.ogv в зависимости от поддерживаемого ими видеоформата.

```
<video controls width="400" height="300">
  <source src="butterfly_mobile.mp4" type="video/mp4"
          media="(max-device-width: 480px)">
  <source src="butterfly.mp4" type="video/mp4">
  <source src="butterfly.ogv" type="video/ogg">
</video>
```

Но, конечно же, ответственность за кодирование файлов в разные форматы лежит на разработчике. Как правило, средства кодирования имеют профили для конкретных устройств (например, "iPad Video"), которые могут облегчить эту задачу. Кроме этого, разработчик также должен удостовериться в применении правильного формата мультимедиа для устройства (обычно это будет формат H.264) и предоставить видеоформаты для всех других браузеров.

Рисование эффектных рамок

С самых ранних дней спецификации CSS веб-дизайнеры пользовались стилями для форматирования рамок с содержимым. По мере того как CSS набиралась сил, предоставляемые ею возможности для оформления рамок становились все более впечатляющими, позволяя создавать разнообразные эффекты от затененных колонтитулов до плавающих рисунков с подписями. А решение в CSS задачи реагирования элементов на наведение курсора превратило плавающие панели в вычурные кнопки и другие элементы с подсветкой, позволив избавиться от неуклюжих прошлых решений на JavaScript. Поэтому неудивительно, что некоторые наиболее популярные и лучше всего поддерживаемые возможности CSS3 могут сделать ваши рамки сногсшибательными независимо от их содержимого.

Прозрачность

Возможность делать изображения и цвета прозрачными является одним из самых фундаментальных строительных блоков в CSS3. Существуют два способа установки прозрачности.

Первый заключается в использовании функции `rgba()`, которая принимает четыре параметра. Первые три параметра задают значения (от 0 до 255) красной, зеленой и синей составляющей цвета. Последний параметр задает значение прозрачности, или *альфа*, которое может быть в диапазоне от 0 (полная прозрачность) до 1 (полная непрозрачность).

Следующее правило задает фон ярко-зеленого цвета 50-процентной прозрачности:

```
.semitransparentBox {
  background: rgba(170,240,0,0.5);
}
```

Браузеры, которые не поддерживают функцию `rgba()`, будут просто игнорировать это правило, и фон сохранит прозрачность по умолчанию — полная прозрачность.

Поэтому второй подход лучше. Здесь мы сначала устанавливаем сплошной резервный цвет, а потом заменяем его полупрозрачным цветом:

```
.semitransparentBox {
  background: rgb(170,240,0);
  background: rgba(170,240,0,0.5);
}
```

Таким образом, браузеры, которые не поддерживают функцию `rgba()`, все равно окрасят фон элемента, но только полностью непрозрачным цветом.

> **Совет**
>
> Чтобы добиться лучшего соответствия резервного цвета с основным, следует использовать такой резервный цвет, который наиболее точно воспроизводит эффект полупрозрачности. Например, если полупрозрачный цвет накладывается на белый фон, то этот цвет будет выглядеть еще светлее из-за просвечивающегося белого фона. При выборе резервного цвета этот эффект следует принять во внимание.

Спецификация CSS3 также определяет свойство стиля `opacity` (непрозрачность), которое работает точно так же, как и значение альфа. Значение `opacity` тоже устанавливается в диапазоне от 0 до 1, позволяя сделать любой элемент полупрозрачным:

```
.semitransparentBox {
  background:  rgb(170,240,0);
  opacity:     0.5;
}
```

На рис. 8.10 показаны два примера полупрозрачности, один из которых реализован с помощью функции `rgba()`, а другой — с помощью свойства `opacity`.



Рис. 8.10. Два разных типа полупрозрачности. Функция `rgba()` применяется, чтобы позволить фону показываться сквозь рамку, но не через текст в рамке *(вверху)*. Свойство `opacity` применяется, чтобы уменьшить интенсивность рисунка, где фон просвечивается сквозь изображение и текст *(внизу)*

Свойство `opacity` предпочтительнее использовать вместо функции `rgba()` в следующих случаях:

- когда нужно сделать полупрозрачными несколько цветов. Свойство `opacity` позволяет сделать полупрозрачными цвет фона, текста и рамки элемента;
- когда нужно сделать что-то полупрозрачным, даже не зная его цвет. Например, потому, что цвет может устанавливаться другой таблицей стилей или кодом сценариев JavaScript;
- когда нужно сделать полупрозрачным изображение;
- когда нужно использовать переход, т. е. эффект анимации, который делает элемент постепенно появляющимся или исчезающим *(см. разд. "Создание эффектов перехода" далее в этой главе)*.

Скругление углов

Мы уже рассмотрели простое свойство border-radius, которое позволяет скруглять углы рамок. Но это свойство можно настроить таким образом, чтобы придавать углам рамок любую кривую форму.

Прежде всего, этому свойству можно присвоить другое, единое значение радиуса границы. Радиус границы — это радиус обрамляющего круга. Конечно же, весь круг не рисуется, а только та его часть, необходимая для соединения вертикальной и горизонтальной границ рамки. Если установить большее значение border-radius, получится больший круг и более плавно скругленный угол. Как и в случае с большинством других измерений в CSS, радиус можно задавать в разных единицах, включая пикселы и проценты. Кроме этого, для каждого угла можно указать отдельное значение border-radius:

```
.roundedBox {
  background:    yellow;
  border-radius: 25px 50px 25px 85px;
}
```

Но это еще не все — также можно растянуть круг в эллипс, создавая кривую, более протяженную в одном из направлений. Для этого нужно обрабатывать каждый угол отдельно специальными свойствами (например, свойство border-top-left-radius

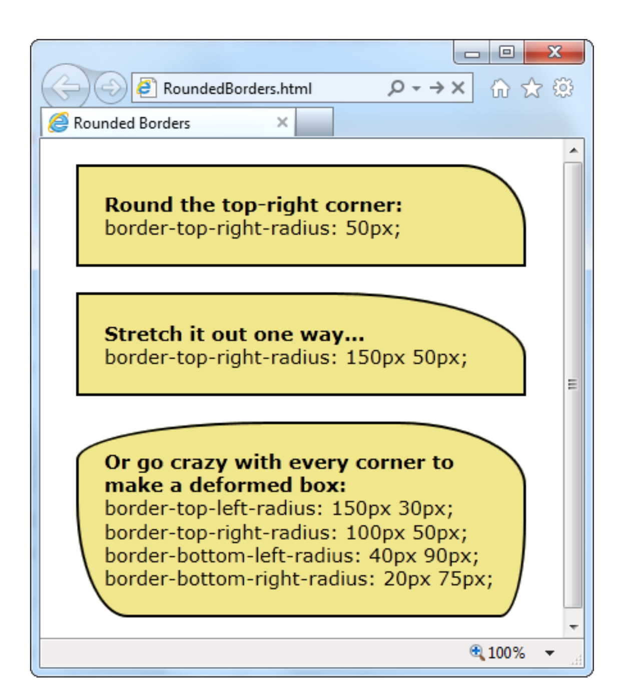


Рис. 8.11. Умелое использование свойства border-radius позволяет скруглять углы рамок практически любым способом

обрабатывает левый верхний угол), которым присваиваются два значения: одно для горизонтального радиуса эллипса, а другое — для вертикального:

```
.roundedBox {
  background:                yellow;
  border-top-left-radius:  150px 30px;
  border-top-right-radius: 150px 30px;
}
```

На рис. 8.11 показано несколько примеров разных типов скругления углов.

Фон

Одним из быстрых способов создания привлекательных фонов и обрамлений будет использование изображения. Спецификация CSS3 определяет две новые возможности, которые можно использовать для этой цели. Первая возможность — это поддержка нескольких фонов, которая позволяет объединить два (или больше) изображения в один фон. Далее приведен пример кода для создания одного фона из двух изображений, чтобы украсить левый верхний и правый нижний углы рамки:

```
.decoratedBox {
  margin:  50px;
  padding: 20px;
  background-image: url('top-left.png'), url('bottom-right.png');
  background-position: left top, right bottom;
  background-repeat:   no-repeat, no-repeat;
}
```

Первым шагом в реализации этой задачи надо предоставить список любого количества изображений в свойстве `background-image`. Полученные изображения можно потом расположить в соответствующих местах посредством свойства `background-position` и указать, повторять ли их, с помощью свойства `background-repeat`. При этом нужно следить за правильностью порядка, чтобы расположить первое изображение в позиции, указанной в первом значении свойства `background-position`, второе — во второй и т. д. Результаты применения правила показаны на рис. 8.12.

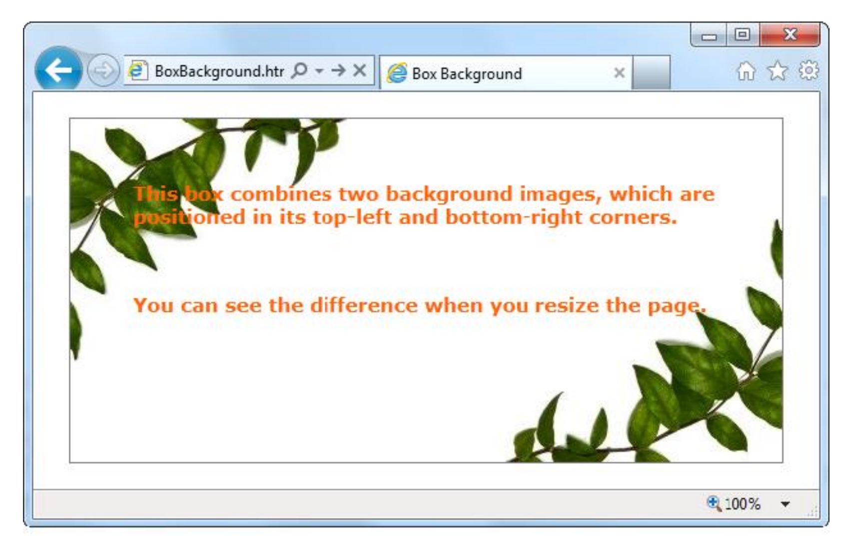
Примечание

Браузеры, не поддерживающие составные фоны, полностью игнорируют попытку установить фон этого типа. Во избежание этой проблемы сначала установите резервный фон, присвоив цвет или изображение свойству `background` или `background-image`. Только потом предпринимайте попытку установить составной фон, присвоив свойству `background-image` список изображений.

А в следующем коде показан пример создания составного фона методом *раздвигаемой двери* (sliding doors). Это испытанный временем шаблон веб-дизайна, в котором графика меняющегося размера создается из трех частей: левого изображения, правого изображения и чрезвычайно узкого изображения посредине:

```
.decoratedBox {
  margin: 50px;
```

```
  padding: 20px;
  background-image: url('left.png'), url('middle.png'), url('right.png');
  background-position: left top, left top, right bottom;
  background-repeat:   no-repeat, repeat-x, no-repeat;
}
```



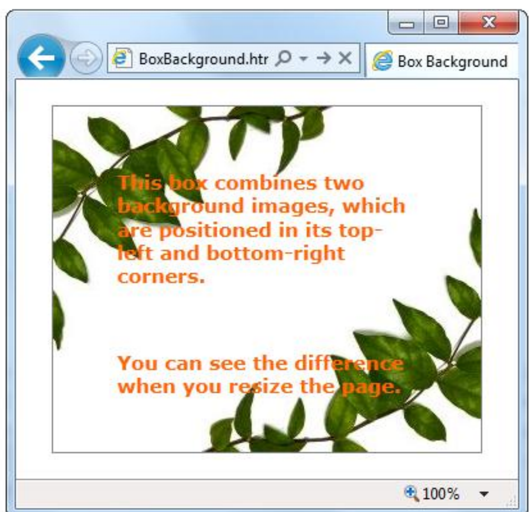


Рис. 8.12. Несмотря на изменения размеров рамки, фоновые изображения остаются в своих углах

Разметку этого типа можно использовать, чтобы создать фон для кнопки. Но, конечно же, имея в багаже все эти замечательные новые возможности CSS3, вы, скорее всего, предпочтете создавать такие объекты, добавляя тени, градиенты и другие эффекты, не использующие изображения. Следующие разделы помогут вам в этом.

Тени

Спецификация CSS3 определяет два новых типа теней: *блочные тени* (box shadows) и *текстовые тени* (text shadows). Блочные тени обычно более полезны и имеют более высокий уровень поддержки, в то время как текстовые тени не работают ни в одной из версий Internet Explorer. Блочную тень можно использовать для создания прямоугольной тени позади любого блока элемента `<div>` (но не забудьте при этом о рамке, чтобы он продолжал выглядеть как блок). Тени могут даже следовать контурам блоков со скругленными углами (рис. 8.13).



Рис. 8.13. С помощью теней можно создавать такие эффекты, как плавающий текст *(вверху)*, выступающий блок *(в центре)* и светящиеся кнопки *(внизу)*

Рассматриваемые тени создаются посредством свойств `box-shadow` и `text-shadow`. Далее приведен пример создания базовой блочной тени:

```
.shadowedBox {
  border: thin #336699 solid;
  border-radius: 25px;
  box-shadow: 5px 5px 10px gray;
}
```

Первые два значения свойства `box-shadow` устанавливают горизонтальное и вертикальное смещения тени от исходного объекта. Положительные значения смещают тень вниз и вправо, отрицательные — вверх и влево. Следующее значение определяет размер *размытия* (blur; в данном примере 10 пикселов), которое увеличивает расплывчатость тени. Последнее значение определяет цвет тени. Если под блоком находится какое-либо содержимое, подумайте об использовании функции `rgba()` *(см. разд. "Прозрачность" главы 6)*, чтобы сделать тень полупрозрачной.

Для более тонкой настройки тени в свойство `box-shadow` можно добавить два значения. Чтобы установить *ширину* (spread) тени — подсвойство, которое расширяет тень, утолщая ее сплошную часть между размытыми краями, добавляется значение между значениями размытия и цвета:

```
box-shadow:  5px 5px 10px 5px gray;
```

А чтобы создать тень, отражающуюся не наружу, а внутрь элемента, в конце списка значений добавляется значение `inset`. Лучший эффект достигается, когда тень располагается непосредственно поверх элемента, без горизонтального или вертикального смещения:

```
box-shadow: 0px 0px 20px lime inset;
```

Таким образом создан нижний пример на рис. 8.13. Вставленные тени можно использовать для создания эффекта подсветки кнопок или рамок при наведении на них курсора мыши *(см. разд. "Простой цветовой переход" далее в этой главе)*.

> **ПРИМЕЧАНИЕ**
>
> Помешанные на тенях разработчики могут даже использовать несколько теней, разделяя их запятыми. Но это обычно напрасная трата усилий и вычислительных возможностей.

Свойство `text-shadow` требует подобного набора значений, но в другом порядке. Сначала указывается цвет, за ним следует горизонтальное и вертикальное смещения, а потом размытие:

```
.textShadow    {
  font-size:   30px;
  font-weight: bold;
  text-shadow: gray 10px 10px 7px;
}
```

Градиенты

Градиенты — это переходы цветов, которые могут создавать широкий диапазон эффектов, от едва различимой тени под панелью меню до психоделически раскрашенных кнопок. На рис. 8.14 показано несколько примеров градиентов.

> **ПРИМЕЧАНИЕ**
>
> На многих веб-страницах градиенты симулируются фоновыми изображениями. Но технология CSS3 позволяет веб-разработчику определить требуемый градиент, который будет воспроизведен браузером. Преимущество этого подхода состоит в том, что он уменьшает количество файлов изображений, которые нужно обрабатывать, и предоставляет возможность создавать градиенты, которые без стыков изменяют свой размер, позволяя заполнять любое пространство.

Мы уже получили определенный опыт работы с градиентами на холсте *(см. разд. "Градиентная заливка фигур" главы 7)*; градиенты CSS3 подобны градиентам холста. Как и холст, CSS поддерживает два типа градиентов: линейные градиенты, в которых переход цветов осуществляется от одной полосы цвета к другой, и радиальные градиенты, в которых переход идет радиально от центра концентрических (но не обязательно) кругов.

В CSS нет никаких специальных свойств для создания градиентов. Вместо них используется функция градиента для установки свойства `background`. Но не забудьте сначала назначить этому свойству сплошной цвет, чтобы создать резервную залив-

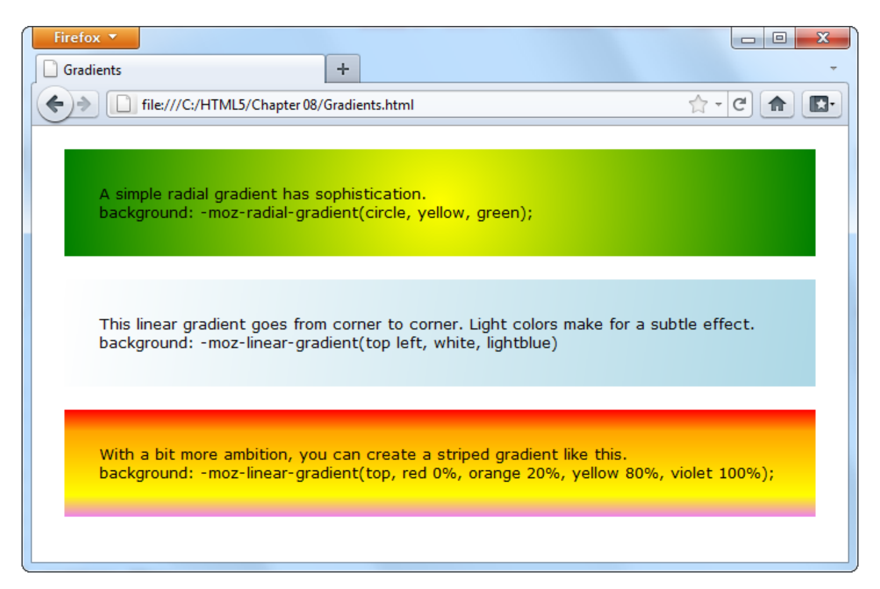


Рис. 8.14. По сути, градиенты — это всего лишь переходы цветов.
Но этот простой рецепт позволяет создавать разнообразные эффекты

ку для браузеров, которые не поддерживают градиенты (включая Internet Explorer, который поддерживает градиенты, только начиная с версии IE 10).

Существуют четыре функции градиентов, и для всех них требуется применять префиксы разработчиков браузеров, о которых мы узнали в *разд. "Стили, специфичные для конкретных браузеров" ранее в этой главе*. В этом разделе мы рассмотрим примеры градиентов для браузера Firefox (для которых применяется префикс `-moz-`). Для поддержки браузеров Chrome, Safari и Opera нужно добавить точно такие же значения градиентов, но с префиксами `-webkit-` и `-o-`.

Первой рассмотрим функцию `linear-gradient()`. Далее приводится одна из ее простейших форм, окрашивающая блок белым цветом вверху, который переходит в синий внизу:

```
.colorBlendBox {
  background: -moz-linear-gradient(top, white, blue);
}
```

Заменив значение `top` на `left`, получим линейный горизонтальный градиент. А указав для начала градиента угол блока, получим диагональный переход:

```
background: -moz-linear-gradient(top left, white, lightblue);
```

Можно создать многоцветный градиент, предоставив список цветов. Например, следующее правило создает трехцветный горизонтальный градиент:

```
background: -moz-linear-gradient(top, red, orange, yellow);
```

Наконец, вместо равномерного распределения цветов градиента можно указать позицию начала каждого цвета посредством *точек остановки градиента* (gradient stops), растягивая или сжимая полосы или смещая их в ту или другую сторону.

Точки остановки градиентов указываются в процентах, где `0%` означает начало градиента, а `100%` — окончание. Следующий пример градиента растягивает оранжево-желтую среднюю часть:

```
background: -moz-linear-gradient(top, red 0%, orange 20%,
                                  yellow 80%, violet 100%);
```

Радиальный градиент создается с помощью функции `radial-gradient()`. Для нее нужно предоставить центр круга и цвет для внешнего края круга, где он совпадает с рамкой элемента. Следующее правило определяет радиальный градиент, начинающийся с белой точки в центре и переходящий к синему цвету на окружности:

```
background: -moz-radial-gradient(circle, white, lightblue)
```

Кроме упомянутых, есть еще много других опций, которые позволяют сместить центр круга, растянуть круг в эллипс, указать точку окончания одного цвета и начало другого и т. п. Но разработчики браузеров все еще пытаются определиться с простым единообразным синтаксисом для градиентов, приемлемым для их всех. С другими примерами градиентов и двумя другими функциями для создания градиентов, не рассмотренными здесь (`repeating-linear-gradient()` и `repeating-radial-gradient()`), можно ознакомиться в блоге, посвященном браузеру Safari, по адресу **www.webkit.org/blog/l424/css3-gradients**. Или же можно попробовать онлайновый инструмент корпорации Microsoft, с помощью которого можно щелчками мыши создать требуемый градиент, а потом получить его разметку для всех браузеров, включая Internet Explorer 10. Находится этот инструмент по этому адресу: **http://tinyurl.com/5rzocsk**.

Совет

Во всех этих примерах градиенты создавались свойством `background`. Но функции градиентов также можно применить и для установки значения свойства `background-image`, что позволяет создать резервное изображение: сначала присваиваем свойству `background-image` соответствующее изображение для менее способных браузеров, а потом присваиваем ему значение посредством градиентной функции. Большинство браузеров достаточно сообразительные и не будут загружать изображение, если оно им не требуется, что позволяет сэкономить на трафике.

Создание эффектов перехода

Добавление в CSS псевдоклассов сделало жизнь легче для всех веб-разработчиков. Теперь они могут создавать интерактивные эффекты с помощью псевдоклассов `:hover` и `:focus`, не прибегая к использованию сценариев JavaScript. Например, чтобы создать меняющуюся кнопку (т. е. кнопку, реагирующую на наведение курсора мыши), достаточно просто предоставить набор новых свойств стиля для псевдокласса `:hover`. Эти свойства задействуются автоматически, когда пользователь наводит курсор на кнопку.

Совет

Если вы последний из веб-разработчиков, кто еще не создавал собственную меняющуюся кнопку, то можете найти информацию по этой теме в подробном руководстве

"Создание Web-сайтов. Основное руководство"[1] или же в статье по адресу **http://www.elated.com/articles/css-rollover-buttons/**.

Какими бы не были псевдоклассы замечательными, они больше не являются передовой технологией. Проблема с ними состоит в их характере типа "все или ничего". Например, настройки стиля псевдокласса `:hover` активируются сразу же при наведении курсора на элемент, т. е. происходит резкий скачок из одного стиля в другой. А вот в приложениях Flash или в прикладных программах эффект обычно более утонченный. Здесь кнопка при наведении курсора может менять цвет, сдвигаться или подсвечиваться, но делается это с использованием тонкой анимации, занимающей долю секунды.

Некоторые веб-разработчики начинают использовать подобные эффекты в своих работах, но для этого обычно требуется прибегать к помощи системы анимации JavaScript от сторонних разработчиков. А теперь CSS3 предоставляет им более простой способ — новую возможность *переходов*, позволяющую осуществлять плавный переход от одного набора стилей к другому.

Простой цветовой переход

Чтобы понять принцип работы переходов, нужно рассмотреть реальный пример. Такой пример показан на рис. 8.15, где изменение цвета кнопки при наведении на нее курсора осуществляется с помощью возможностей перехода CSS3.

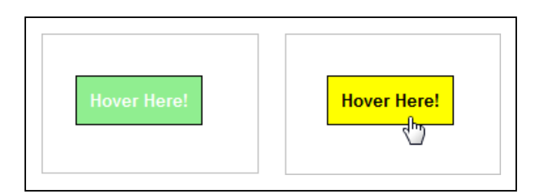


Рис. 8.15. Если бы это была обычная меняющаяся кнопка, при наведении курсора ее цвет бы менялся с зеленого на желтый резким прыжком. Но при использовании эффекта перехода зеленый цвет переходит в желтый плавно, занимая около полсекунды. Таким же образом происходит и обратный переход при отодвигании курсора. В результате получаем более изящную кнопку

Смену цвета без эффекта перехода можно реализовать следующим кодом:

```
.slickButton {
  color:       white;
  font-weight: bold;
  padding:     10px;
  border:      solid 1px black;
  background:  lightgreen;
  cursor:      pointer;
}
```

[1] Мак-Дональд М. Создание Web-сайтов. Основное руководство. — М.: Эксмо, 2010.

```
.slickButton:hover {
  color:      black;
  background: yellow;
}
```

А этот код создает кнопку, которая форматируется только что описанным стилем:

```
<button class="slickButton">Hover Here!</a>
```

Чтобы получить плавное изменение цвета, т. е. переход, нам нужно в только что описанный стиль добавить свойство `transition`. (Обратите внимание, что это свойство вставляется в обычный стиль (в данном случае стиль `slickButton`), а не в псевдокласс `:hover`.)

Как минимум, свойство `transition` требует установки двух значений: свойства CSS, которое нужно анимировать, и времени, на протяжении которого нужно выполнить изменение стилей. В данном примере переход применяется к свойству `background`, а время перехода равно 0,5 секунды:

```
.slickButton {
  color:        white;
  font-weight: bold;
  padding:      10px;
  border:       solid 1px black;
  background:   lightgreen;
  cursor:       pointer;
  -webkit-transition: background 0.5s;
  -moz-transition: background 0.5s;
  -o-transition: background 0.5s;
}

.slickButton:hover {
  color:      black;
  background: yellow;
}
```

Как вы, несомненно, заметили, в предыдущем коде в стиль были добавлены три свойства `transition`, вместо обговариваемого одного. Это потому, что стандарт переходов CSS3 все еще находится в процессе разработки и поддерживающие его браузеры требуют применения префиксов разработчиков. Таким образом, чтобы переход работал в браузерах Chrome, Safari, Firefox и Opera, нужно установить три значения свойства `transition`, добавляя к каждому из них свой префикс разработчика. А для Internet Explorer 10 (который, как ожидается, будет поддерживать переходы) нужно будет добавить еще одну версию свойства, с префиксом `-ms-`. К сожалению, использование экспериментальных свойств может породить неопрятные таблицы стилей.

Так, в данном примере присутствует один мелкий недостаток, а именно, в активированной кнопке меняются два элемента оформления — цвет самой кнопки и цвет текста на ней. Но эффект перехода применяется только к цвету кнопки, в результа-

те чего смена цвета текста происходит практически мгновенно, в то время как цвет кнопки меняется постепенно.

Эту проблему можно решить двумя способами. При первом подходе свойству `transition` присваивается список переходов, разделенных запятыми:

```
.slickButton {
  -webkit-transition: background 0.5s, color 0.5s;
  -moz-transition:    background 0.5s, color 0.5s;
  -o-transition:      background 0.5s, color 0.5s;
}
```

Но этот код можно сократить, если нам нужно установить переход для всех изменяющихся свойств и при одинаковом времени перехода для них всех. В таком случае вместо списка свойств для перехода мы просто используем ключевое слово `all`:

```
-webkit-transition: all 0.5s;
-moz-transition:    all 0.5s;
-o-transition:      all 0.5s;
```

> ***Примечание***
>
> Переход можно настроить более тонко. Прежде всего можно использовать свойство `transition-timing-function`, которое позволяет изменять скорость перехода в процессе его осуществления. Например, начало перехода может быть медленным с последующим ускорением, или наоборот, быстрый переход в начале и замедленный в конце. Для коротких переходов это свойство не играет большой роли. Но в длинных и более сложных анимациях оно может изменить общее восприятие эффекта. Можно также использовать свойство `transition-delay`, которое задерживает начало перехода на указанное время. Дополнительную информацию об этих и других свойствах переходов см. по адресу **www.w3.org/TR/css3-transitions**.

На момент написания этой книги переходы поддерживались браузерами Opera 10.5, Firefox 4, а также всеми версиями Safari и Chrome, с которыми вам когда-либо придется работать. Браузер Internet Explorer переходы не поддерживает, хотя это планируется для версии IE 10. Но отсутствие поддержки переходов — не такая и большая проблема, как может показаться, т. к. браузер все равно меняет стили. Только эта смена происходит почти мгновенно, а не с растянутым во времени переходом. А это уже хорошая новость, т. к. веб-сайт может использовать переходы и в то же самое время сохранять основные визуальные стили для старых браузеров.

Еще несколько идей с переходами

Радует, что предоставляемая CSS возможность переходов способна простой смене цвета придать эстетический вид. Но если вы планируете создать искусный эффект реагирования вашими кнопками и меню при наведении на них курсора, существует много других свойств, с которыми можно применять переход. В частности, можно порекомендовать следующие первоклассные идеи.

- **Прозрачность.** Изменяя свойство `opacity`, можно получить эффект, при котором изображение постепенно исчезает из виду, как бы сливаясь с фоном. Только

помните, что изображение не должно стать полностью прозрачным, т. к. посетитель вашего сайта не будет знать, куда наводить курсор, чтобы оно снова появилось.

- **Тени.** В *разд. "Тени" ранее в этой главе* мы узнали, как с помощью свойства `box-shadow` можно создать тень для любого объекта. Но правильная тень также может создать и хороший эффект при наведении курсора на объект. В частности, рассмотрите использование сильно размытых теней без смещения, что создает более традиционный эффект подсветки.
- **Градиенты.** Измените линейный градиент или создайте радиальный, в любом случае на этот эффект будет трудно не обратить внимание.
- **Трансформации.** Как мы увидим в следующем разделе, посредством трансформаций можно переместить и изменить форму и/или размеры любого элемента. Это свойство делает их идеальным инструментом для переходов.

С другой стороны, как правило, неудачно использовать переходы с такими элементами графического дизайна, как промежутки между рамками элементов и их содержимым, полями и текстом со шрифтами разного размера. Эти операции потребуют больше вычислительных ресурсов (т. к. браузеру требуется выполнить перерасчет компоновки или модификации текста), вследствие чего они выполняются медленно и рывками. Если вам нужно что-то переместить, увеличить или уменьшить, лучше всего использовать трансформацию, как рассматривается в следующем разделе.

Трансформации

Когда мы изучали возможности холста, то затронули возможность трансформаций — способов перемещения, масштабирования, наклона и вращения содержимого. На холсте трансформации можно использовать для изменения рисуемой на нем графики. А трансформации CSS3 служат для изменения внешнего вида элементов. Подобно переходам анимации, трансформации CSS3 являются новой и экспериментальной возможностью. Поэтому при их использовании нужно указывать несколько версий свойства `transform`, каждую со своим префиксом разработчика браузеров. Далее приведен пример правила трансформации для вращения элемента со всем его содержимым:

```
.rotatedElement {
  -moz-transform:    rotate(45deg);
  -webkit-transform: rotate(45deg);
  -o-transform:      rotate(45deg);
}
```

ДЛЯ ТЕХ, КТО ПОНИМАЕТ

Не забывайте о старых браузерах

Как мы уже знаем, браузеры, которые не поддерживают переходы, переключаются от одного стиля к другому мгновенно, что обычно не очень хорошо воспринимается. Кроме этого, старые браузеры также игнорируют возможности CSS3 для изменения сти-

лей, например добавление тени и заливки градиентом кнопки, на которую наведен указатель мыши. Это уже намного хуже, т. к. посетители с такими браузерами *вообще* не смогут наслаждаться эффектами, вызываемыми наведением курсора на элемент.

Эта проблема решается использованием резервного решения, которое старые браузеры понимают. Например, можно создать составное правило для состояния наведенного курсора, которое сначала задает некоторый цвет фона и только *потом* устанавливает градиент. Таким образом, при наведении курсора на элемент старые браузеры смогут увидеть требование изменить цвет фона на некоторый сплошной цвет, а более способные браузеры — правило изменить цвет фона на градиентную заливку. Для еще более продвинутых возможностей можно воспользоваться инструментом Modernizr, который позволит вам определить совершенно иные стили для старых браузеров *(см. разд. "Стратегия 3: добавляйте резервные решения с помощью Modernizr" ранее в этой главе)*.

В предыдущем примере функция `rotate()` вращает элемент на 45° вокруг его центра. Но существует много других функций трансформаций, которые можно применять отдельно или совместно. Например, следующее правило стиля соединяет в цепочку три трансформации. Сначала элемент увеличивается в полтора раза (используя трансформацию `scale`), потом перемещается на 10 пикселов влево (посредством трансформации `scaleX`) и, наконец, наклоняется на 10° (используя трансформацию `skew`):

```
.rotatedElement {
  -moz-transform:     scale(1.5)  scaleX(10px)  skew(10deg);
  -webkit-transform:  scale(1.5)  scaleX(10px)  skew(10deg);
  -o-transform:       scale(1.5)  scaleX(10px)  skew(10deg);
}
```

Примечание

Трансформация `skew` искажает форму элемента. Например, возьмем правильный прямоугольник с закрепленным основанием. Если мы начнем толкать его верхнюю часть в сторону, то она сместится, в то время как основание останется на месте. В результате получим параллелограмм. Дополнительную информацию по трансформациям см. в документации для браузера Firefox по адресу **http://tinyurl.com/6ger2wp**, но не забудьте задать соответствующие префиксы разработчиков при использовании их для других браузеров.

Применение трансформации к какому-либо элементу страницы не влияет на другие ее элементы или ее компоновку. Например, если увеличить элемент, то он просто надвинется на смежные с ним элементы.

Трансформации и переходы хорошо применять вместе. Допустим, что вы хотите создать галерею изображений, наподобие показанной на рис. 8.16.

Разметка этого примера начинается довольно просто, помещая набор изображений в контейнер `<div>`:

```
<div class="gallery">
  <img src="bunny.jpg">
  <img src="cat.jpg">
  <img src="dog.jpg">
  <img src="platypus.jpg">
```

```
  <img src="goose.jpg">
</div>
```

Для элемента <div>, содержащего изображения, применяется следующее правило стиля:

```
.gallery {
  margin:     0px 30px 0px 30px;
  background: #D8EEFE;
  padding:    10px;
}
```

А каждый элемент <img> оформляется следующим стилем:

```
.gallery img {
  margin:     5px;
  padding:    5px;
  width:      75px;
  border:     solid 1px black;
  background: white;
}
```





Рис. 8.16. В данном примере трансформация используется, чтобы выделить изображение, на которое наведен указатель мыши

Обратите внимание, что для всех изображений указаны явные размеры посредством свойства width. Причиной этому является то обстоятельство, что в примере используются изображения слегка большего размера, которые уменьшаются при отображении на странице. Это делается преднамеренно: Таким способом мы удостоверяемся в том, что браузер будет иметь всю требуемую информацию, чтобы увеличить изображение в операции трансформации. Если бы мы пропустили этот шаг и использовали файлы уменьшенных изображений, увеличенная версия была бы расплывчатой.

Теперь перейдем к эффекту, активируемому наведением указателя мыши. Когда пользователь наводит указатель мыши на изображение, активируется последова-

тельность операций трансформации, которые слегка вращают и увеличивают изображение:

```
.gallery img:hover {
  -webkit-transform: scale(2.2) rotate(10deg);
  -moz-transform:    scale(2.2) rotate(10deg);
  -o-transform:      scale(2.2) rotate(10deg);
}
```

Но в данном случае эти преобразования выполняются моментально — сейчас здесь, а потом там. Этот эффект можно сделать более плавным и естественным с помощью перехода, применяемого для обеих операций:

```
.gallery img {
  margin: 5px;
  padding: 5px;
  width: 75px;
  border: solid 1px black;
  -webkit-transition: all 1s;
  -moz-transition:    all 1s;
  -o-transition:      all 1s;
  background: white;
}
```

Теперь процесс вращения и увеличения изображения происходит в течение одной секунды, и столько же времени занимает преобразование в исходное состояние при снятии указателя мыши с изображения.

ПРАКТИЧЕСКИЕ ЗАНЯТИЯ ДЛЯ ОПЫТНЫХ ПОЛЬЗОВАТЕЛЕЙ

Будущее эффектов на основе CSS

Рассмотренные здесь примеры едва затронули возможности, предоставляемые функциями трансформаций и переходов. Хотя разработка этих функций еще далека от завершения, вы можете использовать несколько других экспериментальных возможностей, чтобы расширить их функциональность.

- **Трехмерные (3D) трансформации.** Когда вам надоест перемещать и изменять элементы в двухмерном пространстве, можно попробовать делать это в трехмерном с помощью 3D-трансформаций. Дополнительную информацию по этому вопросу см. здесь: **www.webkit.org/blog/386/3d-transforms**.
- **Анимация.** В настоящее время переходы ограничены довольно простыми взаимодействиями, по большому счету только наведением указателя мыши (псевдокласс `:hover`) и получением фокуса (псевдокласс `:focus`). Возможность анимации расширяет область применения переходов, позволяя применять их динамически, реагируя на событие JavaScript. Например, можно создать эффект вращения элемента, который активируется нажатием кнопки. Документацию по анимации средствами CSS3 см. здесь: **www.w3.org/TR/css3-animations**.
- **Код JavaScript.** Включая и выключая стили посредством кода JavaScript, можно создавать сложные объекты пользовательского интерфейса, например, трехмерную карусель изображений или разворачиваемую группу панелей (часто называемую гармошкой). Ознакомиться с несколькими примерами можно здесь: **http://css3.bradshawenterprises.com**.

Но в настоящее время ни одна из этих возможностей не будет стоить потраченных на ее реализацию усилий. Прежде всего, с ними требуется использовать префиксы разработчиков браузеров, вследствие чего можно очень легко допустить ошибки в коде, что и вызывает необходимость тестирования страницы на всех основных браузерах. Кроме этого, многие из сегодняшних браузеров не поддерживают эти возможности. Например, на момент написания этой книги ни одна из версий Internet Explorer и Opera не поддерживали анимацию. Не поддерживает ее также и Firefox 4. А пытаться разработать обходное решение значит затратить больше усилий, чем на использование другого подхода с самого начала.

В настоящее время лучшим практическим решением для эффектов анимации будет какая-либо библиотека JavaScript, наподобие jQuery UI или MooTools. Но будущее веб-эффектов, несомненно, за технологией CSS3, когда стандарты примут законченную форму и современные браузеры будут установлены на всех компьютерах в мире.



ЧАСТЬ III

Создание интеллектуальных веб-приложений

9

ГЛАВА 9

Хранение данных

В Интернете информацию можно сохранять в двух местах: на веб-сервере и на веб-клиенте (т. е. компьютере посетителя страницы). Определенные типы данных лучше хранить в одном из этих мест, а другие типы — в другом.

Правильным местом для хранения конфиденциальных и важных данных будет веб-сервер. Например, если вы положите какие-либо товары в свою корзину в онлайновом магазине, данные о вашей потенциальной покупке сохраняются на веб-сервере. На вашем компьютере сохраняется лишь несколько байтов данных для отслеживания, содержащих информацию о вас (или, вернее, о вашем компьютере), чтобы веб-сервер знал, какая из корзин ваша. Даже с новыми возможностями HTML5 изменять эту систему нет надобности — она надежная, безопасная и эффективная.

Но хранение данных на сервере не всегда является лучшим подходом, т. к. иногда легче хранить второстепенную информацию на компьютере пользователя. Например, имеет смысл хранить локально *пользовательские настройки* (скажем, параметры, которые определяют способ отображения веб-страницы) и *состояние приложения* (снимок текущего состояния веб-приложения), чтобы посетитель мог продолжить его выполнение с того же самого места позже.

До HTML5 единственным способом локального хранения данных было использование механизма файлов *cookies*, который первоначально был разработан для обмена небольшими объемами идентифицирующей информации между веб-серверами и браузерами. Файлы cookies подходят идеально для хранения небольших объемов данных, но модель JavaScript для работы с ними несколько неуклюжа. Система файлов cookies также вынуждает разработчика возиться со сроками действия и бесполезно пересылать данные туда и обратно по Интернету с каждым запросом страницы.

В HTML5 вводится лучшая альтернативная возможность файлам cookies, которая позволяет легко и просто сохранять информацию на компьютере посетителя. Эта информация может храниться на клиентском компьютере неограниченное время, не отправляется на веб-сервер (если только разработчик не сделает это сам), может быть большого объема и для работы с ней требуется всего лишь пара простых, эффективных объектов JavaScript. Эта возможность называется *веб-хранилищем* и

особенно хорошо подходит для применения с автономным режимом работы веб-сайтов, рассматриваемым в *главе 10,* т. к. позволяет создавать самодостаточные автономные приложения, которые могут сохранять всю требуемую им информацию даже при отсутствии подключения к Интернету.

В этой главе мы подробно исследуем веб-хранилище, а также рассмотрим еще одну, более новую технологию, которая позволяет определенным браузерам считывать содержимое других файлов на жестком диске клиентского компьютера.

Основы веб-хранилища

Функциональность веб-хранилища HTML5 позволяет веб-странице сохранять данные на компьютере посетителя. Эта информация может быть кратковременной, которая удаляется после выключения браузера, или долговременной, которая остается доступной при последующих посещениях веб-страницы.

Примечание

Сохраняемая в веб-хранилище информация в действительности сохраняется не в Интернете, а на компьютере посетителя веб-страницы. Иными, словами, веб-хранилище означает хранение данных не ***в*** Интернете, а хранение данных *из* Интернета.

Существуют два типа веб-хранилищ, которые так или иначе связаны с двумя объектами.

- **Локальное хранилище** использует объект `localStorage` для хранения данных для всего веб-сайта на постоянной основе. Это означает, что если веб-страница сохранит данные в локальном хранилище, эти данные будут доступны для пользователя, когда он возвратится на эту веб-страницу на следующий день, на следующей неделе или в следующем году. Конечно же, большинство браузеров также предоставляет пользователю возможность очистить локальное хранилище. В некоторых браузерах она реализована как стратегия "все или ничего", и посредством ее удаляются все локальные данные, во многом подобно тому, как удаляются cookies-файлы. (В действительности, в некоторых браузерах система cookies и локальное хранилище взаимосвязаны, так что единственным способом удалить локальные данные будет удаление cookies.) А другие браузеры могут предоставлять пользователю возможность просмотра данных для каждого отдельного веб-сайта и удалять данные для выбранного сайта или сайтов.
- **Хранилище данных сеансов** использует объект `sessionStorage` для временного хранения данных для одного окна или вкладки браузера. Эти данные доступны лишь до тех пор, пока пользователь не закроет окно или вкладку, после чего сеанс заканчивается и данные удаляются. Но данные сеанса сохраняются, если пользователь переходит на другой веб-сайт, а потом возвращается обратно при условии, что это происходит в том же окне браузера.

Совет

С точки зрения кода веб-страницы, как локальное хранилище, так и хранилище данных сеансов работают абсолютно одинаково. Разница состоит лишь в длительности хра-

нения данных. Использование локального хранилища предоставляет наилучшую возможность для сохранения требуемой информации для последующих посещений веб-страницы пользователем. А хранилище сеансов служит для хранения данных, которые нужно передавать от одной страницы другой. (В хранилище сеансов можно также хранить временные данные, используемые только на одной странице, но для этой цели прекрасно работают обычные переменные JavaScript.)

Как локальное хранилище, так и хранилище сеансов связано с доменом веб-сайта. Таким образом, если сохранить в локальном хранилище данные для страницы **www.GoatsCanFloat.org/game/zapper.html**, эти данные будут доступны для страницы **www.GoatsCanFbat.org/contact.html**, т. к. обе эти страницы имеют один и тот же домен — **www.GoatsCanFloat.org**. Но эти данные не будут доступны для страниц других доменов.

Кроме этого, т. к. веб-хранилище расположено на компьютере (или мобильном устройстве) данного пользователя, оно связано с этим компьютером, и веб-страница, открытая на данном компьютере и хранящая данные в его локальном хранилище, не имеет доступа к информации, которую она сохранила на другом компьютере. Подобным образом веб-страница создает отдельное локальное хранилище, если вы войдете в систему под другим именем пользователя или запустите другой браузер.

Примечание

Хотя спецификация HTML5 не устанавливает никаких жестких правил в отношении максимального объема хранилища, большинство браузеров ограничивают его 5 Мбайт. В этот объем можно упаковать много данных, но его будет недостаточно, если вы хотите использовать локальное хранилище для оптимизации производительности и кэшировать в нем изображения или видео большого объема (и, по правде говоря, локальное хранилище не предназначено для таких целей). Для хранения большого объема данных все еще развивающийся стандарт базы данных IndexedDB *(см. врезку "Практические занятия для опытных пользователей. В мире баз данных" в конце этой главы)* допускает локальное хранение намного большего объема — обычно 50 Мбайт для начала и больше, по согласию пользователя.

Сохранение данных

Прежде чем поместить фрагмент информации в локальное хранилище или хранилище сеансов, ему необходимо присвоить описательное имя. Это имя называется *ключом* (key) и нужно для того, чтобы данные можно было извлечь в будущем.

Синтаксис для сохранения фрагмента данных следующий:

```
localStorage[keyName] = data;
```

Допустим, например, что нам нужно сохранить фрагмент текста, присвоив ему имя текущего пользователя. Для этих данных мы можем использовать ключ `user_name`:

```
localStorage["user_name"] = "Marky Mark";
```

Конечно же, сохранять фрагмент статического текста не имеет смысла. Как правило, нам требуется сохранять какие-либо переменные данные, например текущую дату, результат математического вычисления или текстовые данные, введенные

пользователем в поля формы. Далее приведен пример сохранения введенных пользователем текстовых данных:

```
// Получаем текстовое поле.
var nameInput = document.getElementById("userName");

// Сохраняем тест, введенный в текстовое поле.
localStorage["user_name"] = nameInput.value;
```

Данные из локального хранилища извлекаются так же легко, как и помещаются туда. Например, следующая строка кода берет сохраненные ранее данные и выводит их в окне сообщений:

```
alert("Вы сохранили: " + localStorage["user_name"]);
```

Этот код возвратит данные независимо от того, когда они были сохранены — пять минут или пять месяцев тому назад.

Конечно же, существует возможность, что данные вообще не были сохранены. Проверить, является ли ячейка хранения пустой, можно так:

```
if (localStorage["user_name"] == null) {
  alert ("Вы еще не ввели свое имя");
}
else {
  // Выводим данные, т. е. имя пользователя, в текстовом поле.
  document.getElementById("userName").value = localStorage["user_name"];
}
```

Сохранение данных в хранилище сеансов осуществляется так же легко. Разница состоит лишь в использовании объекта `sessionStorage` вместо объекта `localStorage`:

```
// Получаем текущую дату.
var today = new Date();

// Сохраняем время в виде текста в формате ЧЧ:мм.
sessionStorage["lastUpdateTime" ] = today.getHours() + ":" +
                                    today.get-Minutes();
```

На рис. 9.1 показана простая тестовая страница, демонстрирующая все изложенное. Испытать эту страницу в действии (и убедиться, что сеансовые данные удаляются после закрытия окна, а локальные данные сохраняются неопределенное время) можно на веб-сайте этой книги (**http://www.prosetech.com/html5/**), запустив файл WebStorage.html.

Примечание

Веб-хранилище также поддерживает менее распространенный синтаксис свойств. Согласно правилам этого синтаксиса, мы обращаемся к ячейке хранения с именем `user_name` как `localStorage.user_name`, а не `localStorage["user_name"]`. Оба типа синтаксиса равнозначны, и использование того или другого является вопросом личного предпочтения.

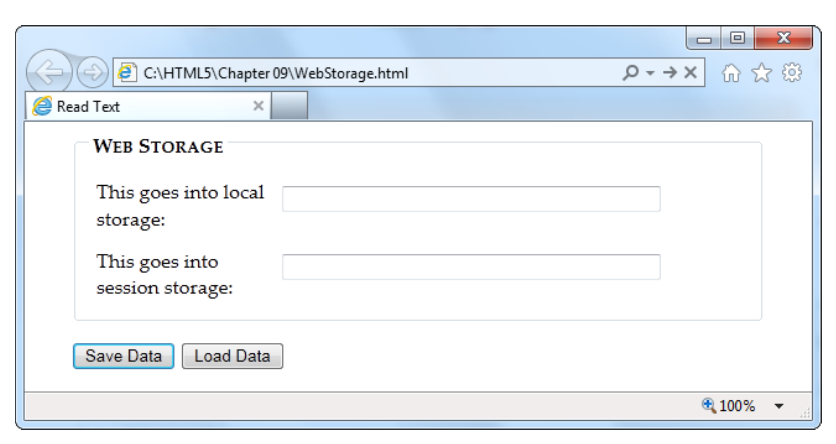


Рис. 9.1. Страница содержит два текстовых поля: для локального хранилища *(вверху)* и для хранилища сеансов *(внизу)*. Нажатие кнопки **Save Data** сохраняет текст, введенный в текстовые поля, а нажатие кнопки **Load Data** выводит в полях соответствующие сохраненные данные

АВАРИЙНАЯ СИТУАЦИЯ
Веб-хранилище не работает без веб-сервера

В своих исследованиях веб-хранилища вы можете столкнуться с неожиданной проблемой. Во многих браузерах веб-хранилище работает только для страниц, предоставленных веб-сервером. При этом не важно, где расположен сервер, в Интернете или на вашем собственном компьютере, самое главное, просто чтобы страницы не запускались с локального жесткого диска (например, двойным щелчком по значку файла страницы).

Эта особенность является побочным эффектом способа, которым браузеры выделяют место в локальное хранилище. Как ранее говорилось, браузеры ограничивают локальное хранилище для каждого веб-сайта 5 Мбайт, для чего им нужно ассоциировать каждую страницу, которая хочет использовать локальное хранилище, с доменом веб-сайта.

Что же происходит, если открыть страницу, которая использует веб-хранилище, с локального жесткого диска? Все зависит от браузера. Браузер Internet Explorer, похоже, полностью утрачивает поддержку веб-хранилища. Объекты `localStorage` и `sessionStorage` исчезают, и попытка использовать их вызывает ошибку JavaScript. В браузере Firefox объекты `localStorage` и `sessionStorage` остаются на месте и, *вроде бы*, поддерживаются (даже Modernizr определяет, что поддерживаются), но все, что отправляется на хранение, исчезает неведомо куда. В браузере Chrome опять же что-то другое — большая часть функциональности веб-хранилища работает как следует, но некоторые возможности (например, событие `onStorage`) не работают. Подобные проблемы возникают и с использованием интерфейса File API *(см. разд. "Чтение файлов" далее в этой главе)*. Поэтому вы избавите себя от многих хлопот, если поместите тестируемую страницу на тестовый сервер, чтобы избежать всех этих неопределенностей. Примеры можно также проверять на веб-сайте этой книги — **http://www.prosetech.com/html5/**.

Практический пример: сохранение текущего состояния игры

На данном этапе у вас может сложиться впечатление, что работа с веб-хранилищем не представляет ничего особенного: нужно всего лишь выбрать ключ для данных и заключить его в квадратные скобки. По большему счету вы будете правы. Но ло-

кальному хранилищу можно дать дополнительное, более практическое применение, не прилагая для этого дополнительных усилий.

Возьмем, например, игру-лабиринт на основе холста, которую мы рассмотрели в *разд. "Практический пример: игра "Лабиринт"" главы 7*. На определенном этапе игры пользователь может отвлечься, но при этом он, вероятно, захочет не просто бросить игру, а сохранить ее текущее состояние, чтобы возобновить ее с этого момента. В таком случае имеет смысл сохранить текущее состояние игры, чтобы пользователь мог возвратиться к ней позже.

Существует несколько способов реализации такой возможности. Можно просто сохранять новую позицию после каждого хода. Поскольку локальное хранилище обладает высоким быстродействием, такой подход вполне реализуем. Либо можно реагировать на событие страницы `BeforeUnload` и запрашивать пользователя, не желает ли он сохранить текущее состояние (рис. 9.2).

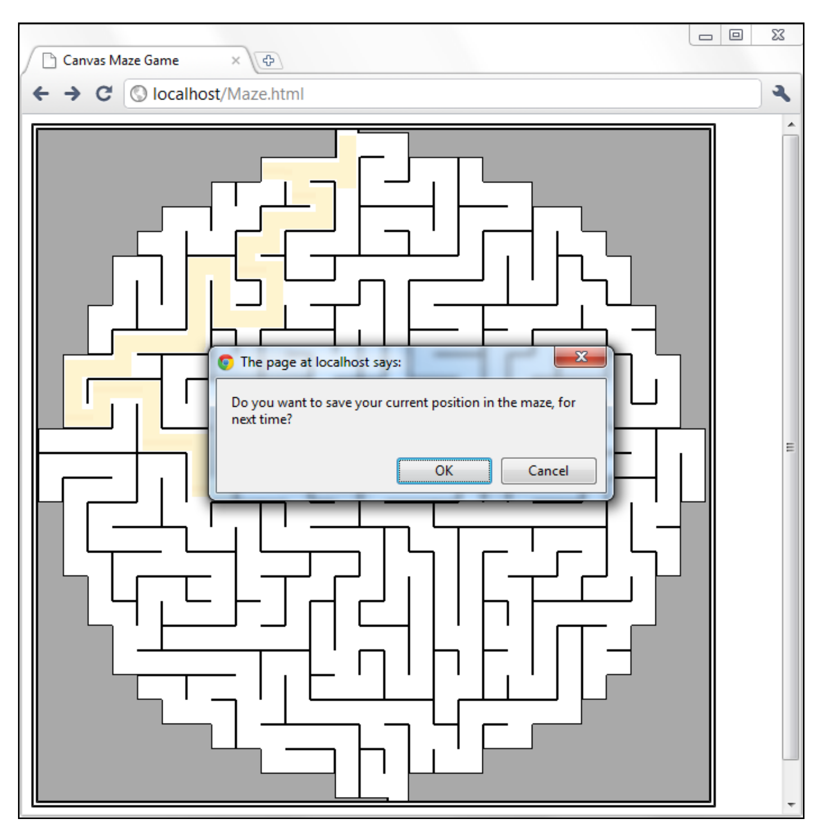


Рис. 9.2. Когда пользователь покидает эту страницу, переходя на новую страницу или закрывая окно, игра предлагает сохранить свое текущее состояние

Код, который предлагает сохранить информацию о состоянии игры, выглядит так:

```
window.onbeforeunload = function(e) {
  // Проверяем, существует ли объект localStorage (т. к. нет смысла
  // предлагать сохранять состояние, если мы не сможем это сделать).
```

```
  if (localStorage) {
    // Выводим запрос о сохранении состояния.
    if (confirm ("Do you want to save your current position
                  in the maze, for next time?")) {
      // Сохраняем две координаты в ячейках хранилища.
      localStorage["mazeCame_currentX"] = x;
      localStorage["mazeCame_currentY"] = y;
    }
  }
}
```

> ***Совет***
>
> Используйте длинный ключ сохраняемых данных наподобие `mazeGame_currentX`. В конце концов разработчик должен обеспечить однозначность ключей, чтобы один и тот же ключ случайно не использовался двумя (или, еще хуже, несколькими) веб-страницами для сохранения разных фрагментов данных. При наличии только одного контейнера для сохранения всех данных породить конфликт ключей проще всего, что является одной из кричащих слабостей системы веб-хранилища. Во избежание подобного рода проблем следует разработать план для создания логичных и описательных ключей. Например, если у вас есть игра "Лабиринт" на нескольких страницах, следует рассмотреть добавление названия страницы в ключ, например `Maze01_currentX`.

При следующей загрузке страницы можно проверить, существует ли эта информация:

```
// Возможность локального хранилища поддерживается?
if (localStorage) {
  // Пытаемся получить данные
  var savedX = localStorage["mazeGame_currentX"];
  var savedY = localStorage["mazeGame_currentY"];

  // При нулевых значениях переменных не сохраняем никаких данных.
  // В противном случае устанавливаем новые координаты
  // по сохраненным данным.
  if (savedX != null) x = Number(savedX);
  if (savedY != null) y = Number(savedY);
}
```

В данном примере показано, как сохранять *состояние приложения*. Если бы мы не хотели надоедать пользователю каждый раз при выходе из игры, можно было бы добавить флажок для автоматического сохранения состояния игры при выходе. При таком подходе состояние игры будет сохраняться автоматически, если этот флажок установлен. Конечно же, здесь нам также нужно было бы сохранять состояние флажка, что дает нам пример сохранения *настроек приложения*.

В этом примере также используется функция `Number()` JavaScript, позволяющая удостовериться в том, что сохраняемые данные преобразованы в действительные числа. Важность этого момента рассматривается в *разд. "Сохранение чисел и дат" далее в этой главе*.

Поддержка веб-хранилища браузерами

Веб-хранилище является одной из наиболее поддерживаемой возможностью HTML5, с хорошим уровнем поддержки в каждом основном браузере. В табл. 9.1 приведены минимальные версии основных браузеров, поддерживающих веб-хранилище.

Таблица 9.1. *Поддержка браузерами локального хранилища и хранилища данных сеансов*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
Минимальная версия	8	3.5	5	4	10.5	2	2

Все эти браузеры предоставляют возможность локального хранилища и хранилища данных сеанса. Но для поддержки события `onStorage` (менее распространенной возможности, которая рассматривается в *разд. "Реагирование на изменения в хранилище" далее в этой главе*) требуются более поздние версии браузеров, например IE 9, Firefox 4 или Chrome 6.

Самой проблемной является версия IE 7, которая не поддерживает веб-хранилище вообще. В качестве обходного решения можно эмулировать веб-хранилище посредством файлов cookies. Это не совсем идеальное решение, но оно работает. Хотя официального сценария для закрытия этого пробела не существует, несколько хороших отправных точек можно найти на странице заполнителей GitHub по адресу **http://tinyurl.com/polyfill** (в разделе "Web Storage").

Продвинутые методы работы с веб-хранилищем

На данном этапе мы познакомились с основами работы с веб-хранилищем — как помещать информацию в него и извлекать эту информацию оттуда. Но нам нужно знать еще несколько подробностей и полезных методов, прежде чем мы сможем применять его. В последующих разделах мы изучим, как удалять информацию из веб-хранилища и просматривать всю находящуюся в нем в данный момент информацию. Также мы рассмотрим, как работать с разными типами данных, сохранять пользовательские объекты и реагировать на изменения в коллекции хранящихся данных.

Удаление элементов

Задача удаления хранящихся в веб-хранилище данных проста до предела. Для удаления отдельного ненужного элемента используется метод `removeItem()`, которому передается соответствующий ключ:

```
localStorage.removeItem("user_name");
```

А если же требуется удалить все локальные данные, сохраненные веб-сайтом, используется более радикальный метод `clear()`:

```
sessionStorage.clear();
```

Поиск всех сохраненных элементов

Чтобы извлечь отдельный элемент данных из веб-хранилища, нам нужно знать его ключ. Но есть еще один ловкий прием. Посредством метода `key()` можно извлечь все элементы, хранящиеся в локальном хранилище или в хранилище сеансов (для текущего веб-сайта), даже если нам не известны их ключи. Этот метод полезен при отладке или когда нужно просто просмотреть, какие данные сохраняют другие страницы веб-сайта и под какими ключами.

На рис. 9.3 показаны результаты применения этого метода.

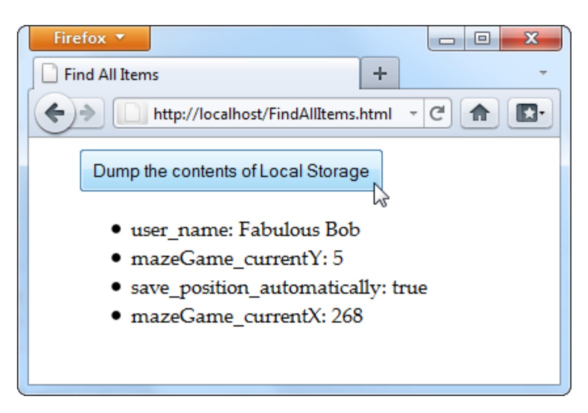


Рис. 9.3. Нажатие кнопки вверху окна выводит содержимое веб-хранилища

В этом примере нажатие кнопки активирует функцию `findAllItems()`, которая сканирует коллекцию элементов в локальном хранилище. Далее приводится полный код примера:

```
function findAllItems() {
  // Получаем элемент <ul> для списка элементов данных.
  var itemList = document.getElementById("itemList");

  // Очищаем список.
  itemList.innerHTML = "";

  // Перебираем все элементы данных в цикле.
  for (var i=0; i<localStorage.length; i++) {
    // Получаем ключ для текущего элемента.
    var key = localStorage.key(i);

    // Получаем элемент, хранящийся под этим ключом.
    var item = localStorage[key];
```

```
    // Создаем с этой информацией новый элемент списка и добавляем
    // его на страницу.
    var newItem = document.createElement("li");
    newItem.innerHTML = key + ": " + item;
    itemList.appendChild(newItem);
  }
}
```

Сохранение чисел и дат

До сих пор в наших исследованиях веб-хранилища мы обходили стороной один важный аспект. А именно — все данные, сохраняемые посредством объектов `localStorage` и `sessionStorage`, автоматически преобразуются в текст.

Со значениями, которые и так являются текстовыми (например, имя пользователя, вводимое в текстовое поле), это обстоятельство не представляет никаких проблем. Но числа не такие сговорчивые. Возвратимся к примеру сохранения состояния игры "Лабиринт" *(см. разд. "Практический пример: сохранение текущего состояния игры" ранее в этой главе)*. Если мы забудем преобразовать координаты обратно в числа, то можем столкнуться со следующей проблемой:

```
// Получаем последнюю позицию координаты x.
// Возвращается, например, текст "35".
x = localStorage["mazeGame_currentX"];

// Увеличиваем координаты.
// К сожалению, JavaScript преобразует "35"+"5" в "355".
x += 5;
```

Это явно не то, что нам требуется. По сути, этот код поместит значок в совершенно неправильное место в лабиринте или даже вне лабиринта вообще.

Здесь проблема состоит в том, что JavaScript полагает, что мы хотим соединить две текстовые строки, а не сложить два числа. Чтобы решить эту проблему, нам нужно дать JavaScript знать, что мы хотим выполнить математическую операцию сложения, а не конкатенацию текста. Это можно сделать несколькими способами, но лучшим будет функция `Number()`:

```
x = Number(localStorage["mazeGame_currentX"]);

// Теперь JavaScript вычисляет 35+10 правильно и возвращает 45.
x += 10;
```

Текст и числа легко поддаются обработке, но при сохранении в веб-хранилище других типов данных следует соблюдать осторожность. Для некоторых типов данных существуют удобные процедуры преобразования. Но допустим, например, что мы сохранили следующую дату:

```
var today = new Date();
```

Этот код сохраняет не объект даты, а текстовую строку, например `Sat Jun 09 2011 13:30:46`. К сожалению, не существует легкого способа для преобразования этого

текста обратно в объект даты при извлечении его из хранилища. А если у нас нет объекта даты, мы не сможем манипулировать этой датой, как и ее исходным объектом, например вызывать его методы и выполнять вычисления.

Чтобы решить эту проблему, разработчик должен явно преобразовать дату в текст, а потом преобразовать извлеченный из хранилища текст обратно в правильный объект даты. Пример таких преобразований приводится в следующем коде:

```
// Создаем новый объект даты.
var today = new Date();

// Преобразуем дату в тестовую строку в формате ГГГГ/ММ/ДД
// и сохраняем эту строку.
sessionStorage["session_started"] = today.getFullYear()
                + "/" + today.getMonth() + "/" + today.getDate();

...

// Теперь извлекаем из хранилища строку даты и с ее помощью
// создаем новый объект даты.
// Это возможно благодаря распознаваемому формату текста даты.
today = new Date(sessionStorage["session_started"]);

// Теперь с этим объектом можно применять его методы.
alert(today.getFullYear());
```

В результате исполнения этого кода выводится окно сообщения, подтверждающее успешное восстановление объекта даты.

Сохранение объектов

В предыдущем разделе мы рассмотрели преобразование чисел и дат в текст для сохранения в веб-хранилище и обратное преобразование данных, извлеченных из веб-хранилища. Выполнение этих преобразований доступно благодаря таким возможностям языка JavaScript, как функция `Number()`, и преобразованиям, зашитым в объекты данных. Но существует большое количество других объектов, которые нельзя преобразовывать таким способом. Классическим примером таких объектов будет пользовательский объект.

Возьмем, например, личностный тест, рассмотренный в *разд. "Обобщая сказанное: рисуем график" главы 7*. Этот тест выполняется на двух страницах. На первой странице пользователь отвечает на вопросы и отправляет их для оценки. На второй странице отображаются результаты теста. В первоначальной версии этой страницы информация передается с первой страницы во вторую с помощью строковых параметров запроса, вставленных в URL. Это традиционный HTML-подход (хотя можно также было бы использовать и cookies). Но в HTML5 лучшим средством для обмена данными является локальное хранилище.

Но здесь есть определенная проблема. Данные теста состоят из пяти чисел, по одному для каждого личностного фактора. Каждое из этих чисел *можно было бы*

сохранить в виде отдельного элемента со своим ключом. Но разве не было бы более аккуратным и компактным создать пользовательский объект, содержащий всю личностную информацию в одном пакете? Далее приведен пример пользовательского объекта `PersonalityScore`, который как раз это и делает:

```
function PersonalityScore(o, c, e, a, n) {
  this.openness = o;
  this.conscientiousness = c;
  this.extraversion = e;
  this.agreeableness = a;
  this.neuroticism = n;
}
```

Использование объекта `PersonalityScore` позволит нам обойтись всего лишь одной ячейкой веб-хранилища вместо пяти. (Чтобы освежить свои знания работы пользовательских объектов в JavaScript, *см. разд. "Отслеживание нарисованного содержимого" главы 7*.)

Но чтобы сохранить пользовательский объект в веб-хранилище, нам нужно преобразовать его в текст. Это можно было бы сделать с помощью кода, что отвечало бы цели, но потребовало бы приложения больших усилий. К счастью, существует более простой, стандартный способ решения этой задачи, который называется *кодированием JSON* (JavaScript Object Notation, система обозначений объектов JavaScript).

Облегченный формат JSON преобразует структурированные данные — наподобие всех значений, обернутых в объект — в текст. Но лучшее в JSON то, что его поддержка встроена в браузеры. Это означает, что мы можем преобразовать в текст любой объект JavaScript вместе с его данными простым вызовом метода `JSON.stringify()`, а метод `JSON.parse()` преобразует этот текст обратно в объект. Далее приведен код примера использования этих методов с объектом `PersonalityScore`. Когда пользователь отправляет свои ответы, страница вычисляет результаты (код вычислений не показан), создает объект, сохраняет его, а потом открывает страницу результатов:

```
// Создаем объект PersonalityScore.
var score = new PersonalityScore(o, c, e, a, n);

// Сохраняем этот объект в формате JSON.
sessionStore["personalityScore"] = JSON.stringify(score);

// Переходим на страницу результатов.
window.location = "PersonalityTest_Score.html";
```

На странице результатов мы можем извлечь из хранилища текст JSON и преобразовать его обратно в требуемый объект с помощью метода `JSON.parse()`. Далее приведен пример кода для такого преобразования и отображения одного из результатов теста:

```
// Преобразуем JSON-текст в соответствующий объект.
var score = JSON.parse(localStorage["personalityScore"]);
```

```
// Извлекаем определенные данные из объекта.
lblScoreInfo.innerHTML = "Your extraversion score is " +
                                                score.extraversion;
```

Полностью код данного примера, включая вычисления каждого личностного фактора, можно просмотреть на сайте книги по адресу **http://www.prosetech.com/html5/**. Узнать больше о системе обозначений JSON и посмотреть, как выглядят данные, закодированные в системе обозначений JSON, можно по адресам **http://ru.wikipedia.org/wiki/JSON** или **http://en.wikipedia.org/wiki/JSON**.

Реагирование на изменения в хранилище

Кроме рассмотренных применений, веб-хранилище предоставляет способ для общения между разными окнами браузера. Это возможно благодаря тому, что каждое изменение локального хранилища или хранилища сеансов активирует событие `window.onStorage` во всех других окнах, в которых просматривается та же страница или другая страница того же самого сайта. Таким образом, изменение локального хранилища для страницы **www.GoatsCanFloat.org/storeStuff.html** активирует событие `onStorage` в окне браузера для страницы **www.GoatsCanFloat.org/checkStorage.html**. (Конечно же, страницы должны просматриваться в том же браузере и на том же компьютере, но вы это уже и так знали, не так ли?)

Событие `onStorage` активируется при добавлении объекта в хранилище, изменении объекта, находящегося в хранилище, удалении объекта из хранилища или полной очистке хранилища. Событие не активируется, если код выполняет операцию с хранилищем, которая не имеет никакого эффекта (например, сохранение значения, которое уже находится в хранилище, или очистка пустого хранилища).

Рассмотрим, например, страницы на рис. 9.4. Здесь в хранилище можно добавить любое значение с любым ключом, заполнив соответствующие текстовые поля и нажав кнопку **Add**. Добавленное значение отображается на другой, уже открытой странице.

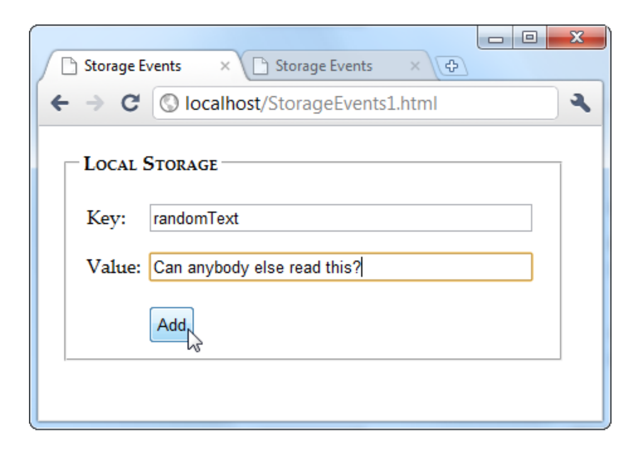
Чтобы создать пример, показанный на рис. 9.4, сначала нужно создать страницу, на которой выполняется ввод и сохранение данных. Данные сохраняются функцией `addValue()`, которая активируется нажатием кнопки **Add** и выглядит так:

```
function addValue() {
  // Получаем значения из обоих текстовых полей.
  var key = document.getElementById("key").value;
  var item = document.getElementById("item").value;

  // Сохраняем элемент в локальном хранилище.
  // (Если ключ уже существует, новый элемент заменяет старый.)
  localStorage[key] = item;
}
```

Вторая страница тоже не представляет ничего сложного. При загрузке страницы событию `window.onStorage` присваивается функция с помощью следующего кода:

```
window.onload = function() {
  // Подключаем событие onStorage к функции storageChanged().
  window.addEventListener("storage", storageChanged, false);
};
```



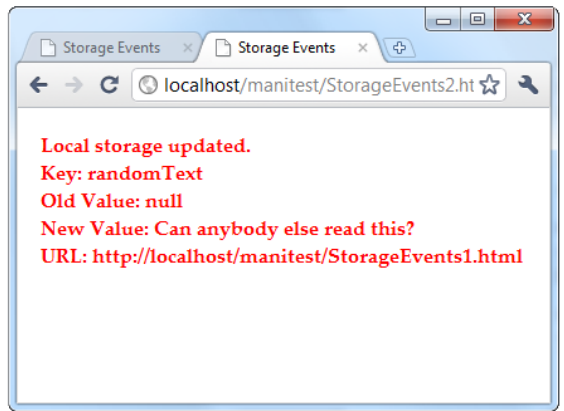


Рис. 9.4. Чтобы увидеть действие события `onStorage`, откройте файлы StorageEvents1.html и StorageEvents2.html. При добавлении или изменении значения на первой странице *(вверху)* вторая страница улавливает это событие и отображает соответствующие данные

Этот код выглядит несколько иначе, чем код для обработки событий, с которым мы сталкивались до сих пор. Вместо установки события `window.onStorage` он вызывает функцию `window.addEventListener()`. Это сделано для того, чтобы код работал на всех современных браузерах. Если активировать событие `window.onStorage` напрямую, код будет работать на всех браузерах, за исключением Firefox.

Примечание

Старожилы Интернета, наверное, помнят, что метод `addEventListener()` не работает на версиях Internet Explorer 8 и более ранних. В данном примере это ограничение несущественно, т. к. IE 8 не поддерживает события хранилища в любом случае.

Функция `storageChanged()` выполняет простую задачу. Она берет обновленную информацию и выводит ее на странице в элементе `<div>`:

```
function storageChanged(e) {
  var message = document.getElementById("updateMessage");
  message.innerHTML = "Local storage updated.";
  message.innerHTML += "<br>Key: " + e.key;
  message.innerHTML += "<br>Old Value: " + e.oldValue;
  message.innerHTML += "<br>New Value: " + e.newValue;
  message.innerHTML += "<br>URL: " + e.url;
}
```

Как видно, событие `onStorage` предоставляет несколько единиц информации, в том числе ключ измененного значения, старое значение, новое значение и URL страницы, на которой изменение было выполнено. Если событие `onStorage` активировано вставкой нового элемента, значение свойства `e.oldValue` равно либо `null` (в большинстве браузеров), либо пустой строке (в Internet Explorer).

ПРИМЕЧАНИЕ

Если открыто несколько страниц одного и того же веб-сайта, событие `onStorage` активируется в каждой из этих страниц, за исключением страницы, на которой было осуществлено изменение (в данном примере это будет страница StorageEvents1.html). Но, как всегда, Internet Explorer не следует этому правилу и активирует событие `onStorage` и на странице, выполняющей изменения.

Чтение файлов

Возможность веб-хранилища поддерживается в HTML5 на хорошем уровне. Но это не единственный способ получения информации веб-страницами. В эту область понемногу вводится несколько других стандартов, направленных на выполнение разных типов задач хранения данных. Одним из таких стандартов является File API, который технически не входит в HTML5, но имеет хороший уровень поддержки на всех современных браузерах (за исключением Internet Explorer).

Судя расплывчатому названию этого стандарта, может показаться, что это всеохватывающий стандарт для чтения и записи файлов на жесткий диск клиента. Но этот стандарт не настолько амбициозный или мощный. Он просто разрешает посетителю веб-сайта выбрать файл на своем жестком диске и передать его непосредственно коду JavaScript, исполняющемуся на просматриваемой веб-странице. Код может открыть этот файл и работать с его данными, будто это простой текст или что-то более сложное. Здесь ключевым аспектом является то обстоятельство, что файл передается непосредственно коду JavaScript. В отличие от обычной выгрузки файла, он никогда не отправляется на веб-сервер.

Также важно знать, что File API *не* может делать. Самое важное, что он не может изменять файлы или создавать новые файлы. Чтобы сохранить какие-либо данные, нужно прибегать к другому механизму, например данные можно отправить на веб-сервер посредством запроса `XMLHttpRequest` *(см. разд. "Объект XMLHttpRequest" главы 11)* или же поместить их в локальное хранилище.

Судя по этому, можно подумать, что интерфейс File API менее полезен, чем локальное хранилище, и для большинства веб-сайтов это будет правильный вывод. Но этот стандарт приоткрывает дверь в область, в которую HTML раньше не входил, по крайней мере без помощи модулей расширений.

Примечание

В настоящее время интерфейс File API является необходимой функциональностью для определенных типов специализированных приложений, но в будущем его возможности могут быть расширены, и важность его значительно возрастет. Например, будущие версии интерфейса могут позволять веб-страницам сохранять файлы на жесткий диск клиента при условии, что пользователь контролирует имя файла и место его сохранения, используя диалоговое окно **Сохранить как**. Модули расширения, наподобие Flash, уже оснащены такой способностью.

Получение файла

Прежде чем интерфейс File API сможет что-либо сделать с файлом, ему нужно этот файл получить. Эту задачу можно выполнить тремя разными способами, но все они одинаковые в одном ключевом аспекте — веб-страница может получить файл только в том случае, если посетитель явно выберет и предоставит его веб-странице.

Способы получения файла следующие.

- **Посредством элемента `<input>`.** Присвоив атрибуту `type` значение `file`, мы получим стандартное окно для закачивания файла. Но с помощью небольшого сценария JavaScript и File API этот файл можно открыть локально.
- **Посредством скрытого элемента `<input>`.** Элемент `<input>` очень непривлекательный. Чтобы не обезображивать им свою страницу, его можно скрыть и создать более прилично выглядящую кнопку. Нажатие этой кнопки активирует JavaScript-код, вызывающий метод `click()` скрытого элемента `<input>`, который открывает стандартное диалоговое окно выбора файла.
- **Посредством метода drag and drop.** Если браузер поддерживает этот метод, файл можно перетащить с рабочего стола или окна браузера и отпустить его в определенной области веб-страницы.

В последующих разделах мы рассмотрим все это подходы более подробно. Но сначала стоит узнать, как обстоят дела с современной поддержкой браузерами интерфейса File API, чтобы вы могли определиться, имеет ли смысл применять этот интерфейс на своем веб-сайте.

Поддержка браузерами интерфейса File API

Интерфейс File API не имеет такой широкой поддержки, как веб-хранилище. Текущая браузерная поддержка этого интерфейса приводится в табл. 9.2.

Все примеры в этой главе могут исполняться на версиях браузеров, перечисленных в табл. 9.2. Но эти браузеры почти наверняка не реализуют все возможности File API, т. к. некоторые части стандарта (для работы с большими объемами двоичных данных и "вырезания" порций данных) все еще находятся в процессе разработки.

Таблица 9.2. *Поддержка браузерами интерфейса File API*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
Минимальная версия	10*	3.6	8	6	11.1	—	3

* На момент написания этой книги доступны только ранние бета-сборки этой версии.

Так как для File API требуются определенные полномочия, которыми обычные веб-страницы не обладают, отсутствующие возможности нельзя заменить дополнительными сценариями JavaScript. Вместо этого нужно применять модули расширения, такие как Flash или Silverlight. Например, на странице **http://sandbox.knarly.com/js/dropfiles** можно найти заполнитель (polyfill), использующий модуль Silverlight, чтобы перехватить перетаскиваемый файл, а потом передать информацию о нем коду JavaScript на странице.

Чтение текстового файла

Самой простой операцией File API будет чтение содержимого простого текстового файла. На рис. 9.5 показана веб-страница, которая с помощью этого интерфейса считывает разметку веб-страницы, а потом выводит этот текст в окне браузера.

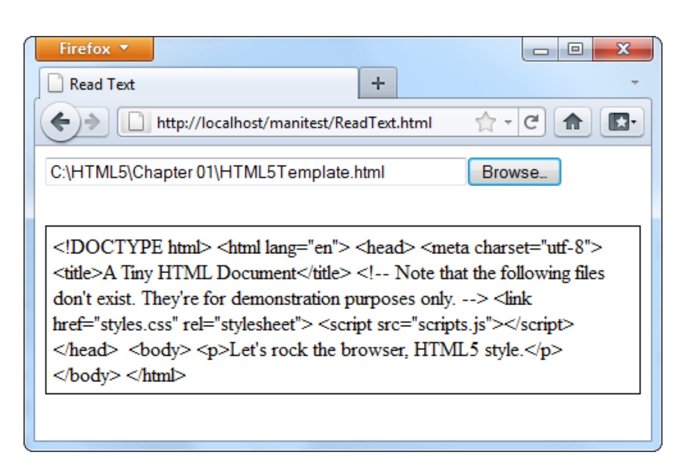


Рис. 9.5. Нажмите кнопку **Browse** (или **Выбрать файл** в Chrome), укажите требуемый файл и нажмите кнопку **OK**. Вместо обычной закачки файла, последующая обработка выполняется кодом JavaScript веб-страницы, который вставляет содержимое в страницу

Создание этого примера начинается с элемента `<input type="file">`, который создает текстовое поле и кнопку **Browse** (Обзор):

```
<input id="filelnput" type="file" onchange="processFiles(this.files)">
```

Но в то время как элемент `<input>` обычно вставляется в контейнер `<form>`, чтобы файл можно было закачать на веб-сервер, в данном случае наш элемент `<input>` играет самостоятельную роль. Когда посетитель страницы выбирает файл, активи-

руется событие `onChange` элемента `<input>`, что в свою очередь активирует функцию `processFiles()`. Как раз на этом этапе и открывается файл посредством самого обыкновенного кода JavaScript.

Теперь рассмотрим по частям функцию `processFiles()`. Сперва нам нужно взять первый файл из коллекции файлов, предоставленных элементом `<input>`. А если только явно не разрешить выбор нескольких файлов (посредством атрибута `multiple`), коллекция файлов будет гарантированно содержать только один файл, размещенный в элементе 0 массива файлов:

```
function processFiles(files) {
  var file = files [0];
```

Примечание

Все объекты файлов обладают тремя потенциально полезными свойствами. Свойство `name` сообщает нам имя файла (без пути), свойство `size` указывает размер файла в байтах, а свойство `type` информирует о MIME-типе файла *(см. врезку "На профессиональном уровне. Основные сведения о MIME-типах" главы 5)*, если его можно определить. Эти свойства можно считывать и использовать их в дополнительной логике, например, отказаться обрабатывать файлы больше определенного размера или разрешить обрабатывать файлы только определенного типа.

Далее создается объект `FileReader` для обработки файла:

```
var reader = new FileReader();
```

Теперь мы почти готовы вызвать один из методов объекта `FileReader`, чтобы извлечь содержимое файла. Но эти методы являются *асинхронными*. Это означает, что они начинают чтение файла, но не ожидают получения данных. Поэтому, чтобы получить данные, сначала нужно обработать событие `onLoad`:

```
reader.onload = function (e) {
  // Когда это событие активируется, данные готовые.
  // Вставляем их в страницу в элемент <div>.
  var output = document.getElementById("fileOutput");
  output.textContent = e.target.result;
};
```

Наконец, подготовив этот обработчик события, можно вызывать метод `readAsText()` объекта `FileReader`:

```
  reader.readAsText(file);
}
```

Этот метод сбрасывает все содержимое файла в одну длинную строку, вставляемую в свойство `e.target.result`, которое в свою очередь отправляется событию `onLoad`.

Метод `readAsText()` работает должным образом только для текстового содержимого файла, но не для двоичного. Это означает, что он идеально подходит для работы с файлами HTML, как показано на рис. 9.5. Но простой текст применяется также в файлах других полезных форматов. Одним из таких форматов является формат CSV — основной формат экспортирования, поддерживаемый всеми программами

электронных таблиц. Другим примером будет формат XML, который является стандартным способом обмена данными между программами. (Этот формат также лежит в основе форматов XML Office, что означает, что docx- и xlsx-файлы можно также передавать напрямую методу `readAsText()`.)

> **ПРИМЕЧАНИЕ**
>
> Язык JavaScript имеет встроенный синтаксический анализатор XML, что позволяет просматривать файлы XML и выбирать из них только требуемое содержимое. Конечно же, для осуществления этой задачи нужно создать код внушительного объема, который плохо работает с большими файлами. К тому же, в редких случаях применение этого кода будет более легким, чем просто загрузка файла на веб-сервер и выполнение обработки там. Но это позволяет продемонстрировать новые возможности, открываемые интерфейсом File API, которые фанаты HTML даже не осмеливались представить себе всего лишь несколько лет тому назад.

Кроме метода `readAsText()`, объект `RileREader` имеет еще несколько других методов для чтения файлов: `readAsBinaryString()`, `readAsDataURL()` и `readAsArrayBuffer()`, но последний метод не поддерживается в Firefox.

Метод `readAsBinary()` предоставляет веб-приложению возможность считывать двоичные данные, хотя он вставляет эти данные в текстовую строку несколько неуклюже, что не является особенно эффективным. А если вы еще захотите разобраться с этими данными, то для этого вам придется мучиться с исключительно запутанным кодом. Лучшая поддержка планируется для возможности "вырезания" небольших фрагментов двоичных данных, чтобы их можно обрабатывать по одному фрагменту за раз. Но на момент написания этих строк эта возможность все еще совершенствуется и реализована по-разному в различных браузерах. С последней версией стандарта можно ознакомиться здесь: **www.w3.orgTR/FileAPI**.

Метод `readAsDataURL()` предоставляет легкий способ захватывать данные изображения. Мы рассмотрим применение этого метода в *разд. "Чтение файла изображения" далее в этой главе*. Но сначала мы выясним, как сделать нашу страницу более красивой.

Замена элемента *<input>*

Веб-разработчики сходятся во мнении: стандартный элемент управления `<input>`, применяемый для выгрузки файлов, выглядит далеко не лучшим образом. Но хотя нам не избежать использования его, совсем необязательно, чтобы пользователи его видели. Мы можем просто скрыть его с помощью следующего стилевого правила:

```
#fileInput {
  display: none;
}
```

Теперь нам нужно добавить новый элемент управления, который будет инициировать процесс предоставления файла. Для этого достаточно обычной кнопки со ссылкой, которую мы можем разукрасить каким угодно образом:

```
<button onclick="showFileInput()">Analyze a File</button>
```

Последним шагом будет обработка нажатия кнопки путем инициализации вручную элемента `<input>` через вызов метода `click()` этого элемента:

```
function showFileInput() {
  var fileInput = document.getElementById("fileInput");
  fileInput.click();
}
```

Теперь нажатие этой кнопки запускает функцию `showFileInput()`, которая "нажимает" скрытую кнопку **Browse** и отображает диалоговое окно для выбора файла. Это, в свою очередь, активирует событие `onChange` скрытого элемента `<input>`, которое запускает функцию `processFiles()` точно таким же образом, как и раньше.

Одновременное считывание нескольких файлов

Нет никаких причин ограничивать количество файлов, предоставляемых пользователем. Стандарт HTML5 разрешает предоставлять для считывания несколько файлов, но это нужно явно указать, вставив атрибут `multiple` в элемент `<input>`:

```
<input id="filelnput" type="file" onchange="processFiles(this.files)"
       multiple>
```

Теперь в диалоговом окне открытия файла пользователь может выбрать несколько файлов (например, удерживая нажатой клавишу <Ctrl> и указывая требуемые файлы или же очертив их при нажатой левой кнопки мыши). Но, разрешив предоставление нескольких файлов, нам нужно и обработать их должным образом. Это означает, что мы не можем просто взять первый файл из коллекции, как мы делали в предыдущем примере. Нам нужен цикл `for`, который обрабатывает все файлы по одному за каждый проход цикла:

```
for (var i=0; i<files.length; i++) {
  // Получаем следующий файл.
  var file = files[i];
  // Создаем объект FileReader для этого файла и исполняем обычный код.
  var reader = new FileReader();
  reader.onload = function (e) {
    ...
  };
  reader.readAsText(file);
}
```

Чтение файла изображения

Как мы узнали, объект `FileReader` обрабатывает текстовое содержимое в один простой прием. Благодаря методу `readAsDataURL()`, он с такой же легкостью обрабатывает и изображения.

На рис. 9.6 показан пример, для реализации которого используются две новые возможности: поддержка изображений и выбор файла методом drag and drop.



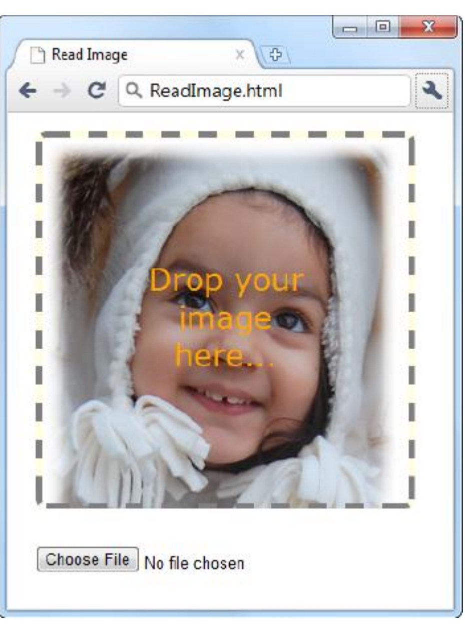


Рис. 9.6. На этой странице изображение можно представить двумя способами: выбрать файл изображения с помощью элементов управления внизу или перетащить и отпустить нужный файл в выделенную пунктиром рамку

Выбранное изображение используется в качестве фона элемента, хотя его с тем же успехом можно было вставить в холст и обрабатывать попиксельно в холсте *(см. разд. "Проверка на столкновение с использованием цвета пикселов" главы 7)*. Этот метод можно применить, например, чтобы создать страницу, на которую посетитель может вставить изображение, доступное для редактирования, а затем сохранить конечный результат посредством вызова объекта `XMLHttpRequest` *(см. разд. "Объект XMLHttpRequest" главы 11)*.

При создании такой страницы сначала нужно решить, какой элемент будет получать перетаскиваемые файлы. В нашем примере это `<div>`-элемент `dropBox`:

```
<div id="dropBox">
  <div>Drop your image here...</div>
</div>
```

С помощью правил таблицы стилей задаем полю для перетаскивания файла желаемый размер и оформляем рамкой и фоном:

```
#dropBox {
  margin: 15px;
  width:  300px;
  height: 300px;
  border: 5px dashed gray;
  border-radius: 8px;
  background: lightyellow;
  background-size: 100%;
```

```
  background-repeat: no-repeat;
  text-align: center;
}

#dropBox div {
  margin:      100px 70px;
  color:       orange;
  font-size:   25px;
  font-family: Verdana, Arial, sans-serif;
}
```

Возможно, вы заметили, что в поле для перетаскивания файла изображения установлены свойства `background-size` и `background-repeat` для подготовки к следующей операции. Когда файл изображения перетаскивается в поле `<div>`, то изображение используется в качестве фона этого элемента. Свойство `background-size` обеспечивает уменьшение размеров изображения, чтобы его можно было видеть полностью. А значение `no-repeat` свойства `background-repeat` обеспечивает, что изображение не повторяется для замащивания оставшегося пространства.

Для обработки перетаскивания и отпускания файла нам требуются три события: `onDragEnter`, `onDragOver` и `onDrop`. При загрузке страницы ко всем этим событиям подключается соответствующий обработчик:

```
var dropBox ;

window.onload = function() {
  dropBox = document.getElementById("dropBox");
  dropBox.ondragenter = ignoreDrag;
  dropBox.ondragover = ignoreDrag;
  dropBox.ondrop = drop;
};
```

Функция `ignoreDrag()` обрабатывает как событие `onDragEnter` (которое инициализируется, когда указатель мыши с перетаскиваемым файлом входит в зону сбрасывания), так и событие `onDragOver` (которое срабатывает постоянно во время движения курсора мыши в зоне сбрасывания). Такой подход возможен потому, что нам никак не нужно реагировать на эти действия, кроме как сообщить браузеру не предпринимать никаких действий. Код функции выглядит следующим образом:

```
function ignoreDrag(e) {
  // Обеспечиваем, чтобы никто другой не получил это событие,
  // т. к. мы выполняем операцию перетаскивания.
  e.stopPropagation();
  e.preventDefault();
}
```

Событие `onDrop` более важное, т. к. в нем мы получаем файл и обрабатываем его. Но поскольку файл для страницы можно предоставить двумя способами, собственно для выполнения работы функция `drop()` вызывает функцию `processFiles()`:

```
function drop(e) {
  // Аннулируем это событие для всех других.
  e.stopPropagation();
  e.preventDefault();

  // Получаем перемещенные файлы.
  var data = e.dataTransfer;
  var files = data.files;
  // Передаем полученный файл функции для обработки файлов.
  process Files(files);
}
```

Функция `processFiles()` является последним этапом в процессе перетаскивания файла. Она создает объект `FileReader`, подключает функцию обработки к событию `onload` и вызывает метод `readAsDataURL()` для преобразования данных изображения в данные URL *(см. разд. "Сохранение содержимого холста" главы 6)*:

```
function processFiles(files) {
  var file = files[0];

  // Создаем объект FileReader.
  var reader = new FileReader();

  // Указываем объекту, что делать, когда URL данных готов.
  reader.onload = function (e) {
    // Используем URL изображения для заполнения фона
    // зоны сбрасывания файла изображения.
    dropBox.style.backgroundImage = "url('" + e.target.result + "')";
  };

  // Начинаем считывать изображение.
  reader.readAsDataURL(file);
}
```

Примечание

Как мы узнали в процессе изучения холста, URL данных — это способ представления данных изображения в виде длинной текстовой строки, которую можно использовать как URL. Это позволяет нам перемещать данные изображения из одного места в другое. Чтобы показать изображение на веб-странице, URL данных изображения присваивается как значение свойству `src` элемента `<img>` *(см. разд. "Сохранение содержимого холста" главы 6)* или же как значение CSS-свойства `background-image` (как делается в этом примере).

Объект `FileReader` имеет еще несколько других событий, которые можно использовать при чтении файлов изображений. Событие `onProgress` срабатывает периодически в процессе длинных операций, чтобы предоставить информацию об объеме загруженных данных на текущий момент. (Операцию можно аннулировать до ее завершения, вызвав метод `abort()` объекта `FileReader`.) Событие `onError` срабатывает в случае проблем с открытием или чтением файла. А событие `onLoadEnd` — при за-

вершении операции любым способом, включая ее преждевременное завершение вследствие ошибки.

ПРАКТИЧЕСКИЕ ЗАНЯТИЯ ДЛЯ ОПЫТНЫХ ПОЛЬЗОВАТЕЛЕЙ

В мире баз данных

Вы жаждете более мощной возможности локального хранения данных? Если веб-хранилище (текстовые строки) и File API (простые файлы) вам больше неинтересны, то как насчет полной, миниатюрной базы данных, работающей прямо в браузере?

Эта амбициозная идея не является необоснованной мечтой и уже была однажды реализована в браузерах Chrome и Safari в соответствии со стандартом Web Databases. Но для разработчиков браузера Firefox этот стандарт не был достаточно убедительным, и теперь он считается устаревшим. Вместо него сейчас используется стандарт IndexedDB, который, скорее всего, реализуют все разработчики браузеров. Ознакомиться с этим стандартом можно в документации Firefox по адресу **http://developer.mozilla.org/en/IndexedDB** или же испробовать экспериментальный прототип для Internet Explorer 9, который можно загрузить отсюда: **http://tinyurl.com/3fuvu9g**.

10

ГЛАВА 10

Автономные приложения

Чтобы просматривать веб-сайт, нужно подключиться к Интернету. На сегодняшний день все это знают. Так зачем же тогда нужно рассматривать автономные приложения? Как будто мы собираемся вернуться прошлое столетие. В конце концов, разве на своем пути к мировому господству веб-приложения не сбросили владычество нескольких поколений автономных настольных приложений? Осуществление множества разнообразных задач — от записей в социальной сети до покупки книг в интернет-магазине — просто невозможно без подключения к Интернету. Но не стоит забывать, что даже веб-приложения не предназначены для *обязательного постоянного* пребывания в Сети. В них заложена возможность продолжать работу в периоды временного отсутствия подключения к Интернету. Иными словами, автономное веб-приложение допускает временные сетевые отказы.

Это обстоятельство имеет особо важное значение для посетителей, пользующихся для доступа к сети портативными устройствами, такими как смартфоны и планшетные компьютеры с возможностью веб-доступа. Чтобы понять суть данной проблемы, представьте себе, что вы работаете с веб-приложением на одном из таких устройств, находясь в поезде, и в это время поезд въезжает в туннель. Шансы велики, что ваше веб-приложение выдаст сообщение о неисправимом сбое и вам придется начать все заново, когда поезд выедет из туннеля и восстановится подключение к Интернету. А вот *автономное* веб-приложение позволит избежать такого неблагоприятного развития событий. Хотя некоторые возможности могут стать временно недоступными, приложение в основном останется работоспособным. (Опять же, некоторые туннели длиннее других. Амбициозное автономное приложение может продолжать работу на протяжении трехчасового авиарейса или трехнедельного путешествия по Африке, если это то, что вам нужно. По сути, время нахождения в автономном режиме ничем не ограничено.)

В этой главе мы узнаем, как любую веб-страницу (или группу веб-страниц) можно преобразовать в автономное приложение. Также мы рассмотрим, как определить, когда веб-сервер является доступным, а когда нет, и реагировать соответствующим образом.

ЧАСТО ЗАДАВАЕМЫЙ ВОПРОС
Когда работа в автономном режиме оправдана?

Следует ли мне снабдить свою страницу возможностью работы в автономном режиме?

Автономный режим работы не подходит для всех без разбору типов веб-страниц. Например, по большому счету нет смысла предоставлять автономные возможности для страницы биржевых котировок, т. к. вся суть такой страницы состоит в получении свежих биржевых котировок с веб-сервера. Но возможность работы в автономном режиме может подойти для страницы для анализа акций, которая загружает партию данных, а потом предоставляет возможность анализировать их и строить по ним графики. С такой страницей можно загрузить партию данных, пока существует интернет-подключение, а потом обрабатывать их, не беспокоясь, есть оно или нет.

Автономный режим работы также подходит для интерактивных приложений, хранящих состояние, которые задействуют страницы кода JavaScript для сохранения в памяти информации о своем состоянии. Такие страницы выполняют большой объем работы самостоятельно, поэтому лучше оснастить их возможностью работы в автономном режиме. Кроме этого, стоимость потери подключения к таким страницам также выше, т. к. это может случиться во время сложных и длительных вычислений, которые придется выполнять снова после восстановления подключения.

Другим фактором, который нужно принять во внимание, являются посетители вашего сайта. Работа в автономном режиме нужна, если среди ваших посетителей есть такие, у которых нет надежного подключения к Интернету или которым требуется мобильное подключение (например, если вы создаете картографический инструмент для мобильных устройств).

Однако нужно быть готовым и к тому, что организация работы в автономном режиме окажется несоизмеримой с затраченными на ее реализацию усилиями.

Кэширование файлов с помощью манифеста

Основным техническим приемом, который делает автономный режим работы возможным, является кэширование — загрузка файла с веб-сервера и сохранение его копии на жестком диске клиента. Таким образом, при потере интернет-подключения браузер сможет использовать копию страницы, сохраненную в кэше.

Для создания страницы с возможностью работы в автономном режиме нужно выполнить следующие три шага:

1. Создать файл манифеста. *Манифест* — это специальный файл, в котором хранится информация, указывающая браузерам, какие файлы следует сохранять, какие не сохранять, а какие файлы заменять каким-либо другим содержимым. Этот пакет кэшируемого содержимого называется *автономным приложением* (offline application).
2. Модифицировать веб-страницу, чтобы она обращалась к манифесту. Таким образом, браузер будет знать, что при запросе страницы нужно загрузить файл манифеста.
3. Настроить веб-сервер. Самое важное — веб-сервер должен предоставлять файл манифеста с правильным MIME-типом. Но, как мы увидим далее, есть еще не-

сколько тонких аспектов, которые могут отрицательно повлиять на кэширование страницы.

Способы решения всех этих задач мы рассмотрим в последующих разделах.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Традиционное кэширование и автономные приложения

В области веб-разработки кэширование не представляет собой ничего нового. Браузеры регулярно кэшируют содержимое, чтобы не загружать повторно одни и те же файлы. В конце концов, если несколько страниц веб-сайта используют один и тот же файл таблицы стилей, зачем загружать его для каждой страницы? Но механизм управления этим типом кэширования отличается от механизма работы автономных приложений.

Традиционное кэширование выполняется, когда вместе с запрошенным браузером файлом веб-сервер отправляет дополнительную информацию, которая называется *заголовком Cache-Control*. Эти заголовки предоставляют браузеру информацию о том, следует ли кэшировать данный файл и на протяжении какого времени использовать кэшированную копию, прежде чем запрашивать у веб-сервера, не изменился ли этот файл. Обычно, кэшированная копия содержимого веб-страниц хранится короткое время; намного дольше хранятся кэшированные копии ресурсов, используемых веб-страницами, таких как таблицы стилей, изображения и файлы сценариев.

В противовес, автономное приложение управляется отдельным файлом — манифестом, и длительность его хранения не управляется временными лимитами. Вместо временных лимитов используется следующее правило: если веб-страница является частью автономного приложения, если имеется кэшированная копия этого приложения и если определение этого приложения не претерпело никаких изменений, тогда используется кэшированная копия.

Веб-разработчик может добавить к этому правилу определенные исключения и дополнения, например, указать браузеру не кэшировать определенные файлы или заменить один файл каким-либо другим. Но ему нет надобности заботиться о сроках хранения и других, потенциально проблематичных подробностях.

Создание манифеста

Манифест является центральной частью предоставления автономной работы HTML5. Это текстовый файл, содержащий список файлов, которые нужно кэшировать.

Файл манифеста всегда начинается словами (прописными буквами):

```
CACHE  MANIFEST
```

После этого заголовка следует список файлов, которые нужно кэшировать. Далее приведен пример файла манифеста для кэширования двух веб-страниц (из приложения личностного теста в *разд. "Обобщая сказанное: рисуем график" главы 7*):

```
CACHE  MANIFEST

PersonalityTest.html
PersonalityTest_Score.html
```

Интервалы (как пустая строка в приведенном примере манифеста) не обязательны, можете вставлять их в любом месте.

Для автономного приложения браузер должен кэшировать все, что может потребоваться этому приложению: веб-страницы и ресурсы, используемые этими веб-страницами (например, сценарии, графику, таблицы стилей и встроенные шрифты). Далее приведен пример более подробного манифеста со всеми типами файлов:

```
CACHE  MANIFEST
# pages
PersonalityTest.html
PersonalityTest_Score.html

# styles & scripts
PersonalityTest.css
PersonalityTest.js

# pictures & fonts
Images/emotional_bear.jpg
Fonts/museo_slab_500-webfont.eot
Fonts/museo_slab_500-webfont.woff
Fonts/museo_slab_500-webfont.ttf
Fonts/museo_slab_500-webfont.svg
```

Обратите внимание на две новые детали в этом манифесте. Первая — несколько строчек начинаются символом `#`. Эти строчки являются комментариями, предоставляющими информацию относительно типа следующего за ними содержимого. Вторая — для некоторых файлов указаны пути, например Images/emotional_bear.jpg. При условии, что эти файлы находятся на веб-сервере и браузер может иметь доступ к ним, их можно включать в пакет автономного приложения.

Для сложных веб-страниц потребуется большое количество вспомогательных файлов, что может порождать длинные и сложные файлы манифеста. Но наихудшее — это то, что одна-единственная ошибка в имени файла вызовет полный сбой автономного приложения. В перспективе основные веб-редакторы должны облегчить жизнь веб-разработчиков в этом отношении. В них будут добавлены возможности автоматического создания файлов манифеста для указанных файлов и средства для их модифицирования и сопровождения.

Совет

В некоторых случаях разработчик может не включать в автономное приложение большие или несущественные ресурсы, такие как, например, изображения большого размера или рекламные баннеры. В этом нет ничего предосудительного, но если есть подозрения, что отсутствие таких файлов способно вызвать какие-либо проблемы (например, сообщения об ошибках, странные пустые области или искаженную компоновку страницы), рассмотрите возможность применения JavaScript для настройки ваших страниц в автономном режиме с помощью метода проверки подключения *(см. разд. "Проверка подключения" далее в этой главе)*.

Созданный файл манифеста сохраняется в корневой папке веб-сайта. Ему можно присвоить любое имя, но в настоящее время наиболее популярными расширениями файла являются manifest и appcache. Первое расширение (например,

PersonalityTest.manifest) было бы наиболее логичным, но оно конфликтует с расширением, используемым на некоторых серверах Windows (как часть процесса развертывания ClickOnce, используемого приложениями .NET). Второе расширение файла (т. е. PersonalityTest.appcache) тоже логично, но менее популярно. В любом случае, главное — нужно настроить веб-сервер, чтобы он распознавал используемое расширение файла манифеста. Если у вас свой веб-сервер, вы можете это сделать, пользуясь инструкцией, изложенной в *разд. "Помещение манифеста на веб-сервер" далее в этой главе*. В противном случае вам нужно связаться с администратором вашего хостинга и узнать, какие расширения файлов используются для поддержки файлов манифеста.

ЧАСТО ЗАДАВАЕМЫЙ ВОПРОС
Каким может быть объем кэша?

Есть ли какие-либо ограничения на объем кэшируемой информации?

Максимальный объем кэша, выделяемого для автономных приложений, варьируется, в зависимости от браузера.

Очевидно, что браузеры на мобильных устройствах находятся на нижнем пределе диапазона. Так как объем хранилища на таких устройствах ограничен, им приходится быть экономными в своем кэшировании. На момент написания этих строк версия браузера Safari для iPad и iPhone ограничивала каждое автономное приложение 5 Мбайт кэша.

Браузеры настольных компьютеров на удивление неоднородны. Firefox по умолчанию выделяет каждому автономному приложению 50 Мбайт кэша, а пользователь может еще повысить этот предел. (Пользовательские параметры кэша в Firefox можно установить на вкладке **Network** в окне, открываемом после выбора последовательности команд меню **Tools | Options | Advanced**.) А вот браузер Chrome выделяет каждому автономному приложению ничтожные 5 Мбайт кэша, обойти которое можно либо с помощью специализированного приложения Chrome (см. **http://code.google.com/chrome/extensions/apps.html**), либо изменением настроек браузера (см. **http://tinyurl.com/5w83opp**). Разработчики браузера Chrome планируют в будущем устранить это ограничение и предоставить пользователям явный контроль над параметрами кэша для каждого автономного приложения, но на данном этапе 5 Мбайт — это все, что они получают.

К сожалению, этот неоднородный подход к выделению кэша браузерами представляет проблему. Например, автономное приложение, требующее больше чем 5 Мбайт кэша, будет работать без проблем в браузере Firefox, но не в Chrome. Что еще хуже, при посещении вашего сайта пользователем Chrome этот браузер будет пытаться кэшировать приложение, не сможет этого сделать по причине ограничения, после чего просто выбросит все загруженные данные. Это не только тратит впустую время и трафик, но также не предоставляет пользователю никакой возможности работать в автономном режиме. Он просто не сможет использовать ваше приложение до тех пор, пока не подключится к Интернету.

Из всего этого можно сделать следующий вывод: в перспективе автономные приложение будут, скорее всего, иметь большие объемы кэша в своем распоряжении. Но в настоящее время веб-разработчики должны учитывать, что им доступен намного меньший объем — всего 5 Мбайт. Вследствие этого непрактично использовать возможность автономной работы для улучшения производительности, например, загружая большие, часто используемые файлы и сохраняя их в течение длительного времени. Но хочется надеяться, что эта ситуация в скором времени изменится.

Использование манифеста

Самого факта создания манифеста недостаточно, чтобы браузер обращал на него внимание. Чтобы манифест имел какой-либо эффект, на него должны существовать ссылки в веб-страницах. Для этого нужно вставить в корневой элемент `<html>` атрибут `manifest` и присвоить ему в качестве значения имя файла манифеста:

```
<!DOCTYPE  html>
<html lang="en" manifest="PersonalityTest.manifest">
...
```

Это нужно проделать для *каждой страницы*, входящей в автономное приложение. Для предыдущего примера это означает, что таким способом нужно модифицировать два файла: PersonalityTest.html и PersonalityTest_Score.html.

> **Примечание**
>
> Веб-сайт может иметь неограниченное количество автономных приложений при условии, что каждое из них имеет собственный манифест. Автономные приложения также могут совместно использовать некоторые ресурсы (например, таблицы стилей), но каждое приложение должно иметь свой отдельный набор веб-страниц.

Помещение манифеста на веб-сервер

Тестирование файла манифеста может оказаться сложным процессом. Небольшие ошибки могут вызывать скрытые сбои и нарушать весь процесс кэширования. Но, тем не менее, на определенном этапе файл манифеста нужно протестировать, чтобы удостовериться в том, что разработанное автономное приложение работает так, как вы ожидаете.

Не должно быть неожиданностью то обстоятельство, что автономное приложение нельзя тестировать, запуская файлы на исполнение с локального жесткого диска. Вместо этого автономное приложение нужно разместить на веб-сервере, будь то веб-сервер поставщика услуг Интернета или тестовый веб-сервер разработчика, например веб-сервер ISS, встроенный в Windows.

Тестирование автономного приложения выполняется в такой последовательности:

1. Первым делом необходимо удостовериться в том, что для предоставления файлов манифеста в настройках веб-сервера указан MIME-тип `text/cache-manifest`. Если веб-сервер будет указывать какой-либо другой тип файла, включая простой текстовый файл, браузер будет полностью игнорировать манифест.

> **Примечание**
>
> Настройка веб-серверов выполняется по-разному. В зависимости от уровня вашего опыта и навыков вам может потребоваться помощь знакомого веб-мастера для установки MIME-типа (см. шаг 1) и изменения параметров кэширования (см. шаг 2). Дополнительную информацию о MIME-типах *см. во врезке "На профессиональном уровне. Основные сведения о MIME-типах" главы 5*.

2. Рассмотрите отключение обычного кэширования *(см. врезку "На профессиональном уровне. Традиционное кэширование и автономные приложения" ранее*

в этой главе) для файлов манифеста. Это может быть необходимым по той причине, что веб-сервер может давать указание браузерам кэшировать файлы манифеста в течение определенного времени, точно так же, как и другие типы файлов. Такое поведение логично, но оно может вызвать большие проблемы. Проблема состоит в том, что некоторые браузеры могут игнорировать обновленные файлы манифеста на веб-сервере и продолжать использовать старые, кэшированные файлы манифеста, вследствие чего они также будут продолжать использовать старые, кэшированные файлы веб-страниц. (Особенно грешит неохотой расставаться со старыми файлами манифеста браузер Firefox.) Во избежание этой проблемы веб-сервер необходимо настроить так, чтобы тот не указывал браузерам кэшировать файлы манифеста.

3. Откройте страницу в браузере, поддерживающем автономные приложения, иными словами, практически в любом браузере, за исключением Internet Explorer (см. табл. 10.1). Когда браузер обнаружит, что веб-страница использует манифест, он может запросить разрешение на загрузку файлов. Вероятнее всего, что такое разрешение будут требовать браузеры мобильных устройств по причине ограниченных возможностей хранения. Браузеры настольных компьютеров могут либо выдавать такой запрос, либо нет. Например, Firefox запрашивает разрешение на кэширование (рис. 10.1), а Chrome и Safari — нет.

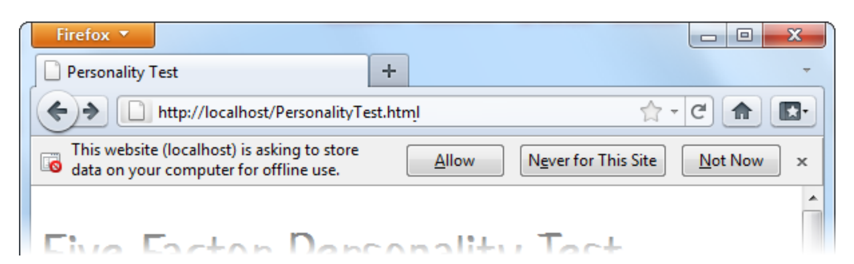


Рис. 10.1. При загрузке страницы, которая использует манифест, браузер Firefox запрашивает разрешения на кэширование файлов. Чтобы разрешить загрузку и кэширование указанных в манифесте файлов, нужно нажать кнопку **Allow**. Если при последующих посещениях страницы Firefox обнаружит измененный файл манифеста, он загрузит новые файлы, не спрашивая разрешения

Получив разрешение, браузер начинает кэширование файлов. Сначала загружается манифест, а потом все перечисленные в нем файлы. Процесс загрузки осуществляется в фоновом режиме, позволяя, таким образом, продолжать просмотр страницы. Этот процесс подобен тому, когда браузер загружает изображение или видео большого объема и при этом отображает остальную страницу.

4. Отключитесь от Интернета. Если ваше приложение размещено на удаленном сервере, просто разорвите подключение. Если же приложение размещено на локальном сервере (т. е. на том же компьютере, что и браузер), остановите веб-сайт (рис. 10.2).

5. Перейдите на одну из страниц автономного приложения и обновите ее. Даже если дать указание браузеру не кэшировать страницу, иногда он все равно сохранит ее, чтобы можно было возвращаться на ранее просмотренные страницы, на-

жимая кнопку **Назад**. Но если обновить страницу, браузер всегда будет пытаться подключиться к веб-серверу. При запросе обыкновенной страницы и при отсутствии подключения к Интернету этот запрос не выполнится. Но при обновлении страницы автономного приложения браузер незаметно предоставляет копию страницы из кэша, даже не информируя пользователя об этом. Можно переходить с одной страницы автономного приложения на другую, но если перейти по ссылке на страницу, не являющуюся частью автономного приложения, браузер выдаст знакомое сообщение о невозможности подключиться к удаленному серверу.

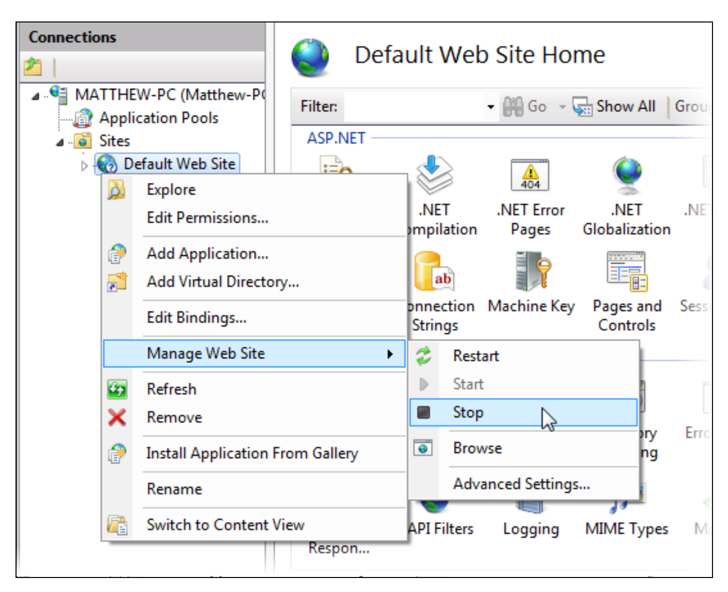


Рис. 10.2. Точный способ остановки веб-сайта зависит от используемого веб-сервера. Здесь показан процесс остановки веб-сайта на сервере IIS для Windows

АВАРИЙНАЯ СИТУАЦИЯ

Мое автономное приложение не работает в автономном режиме

Автономные приложения несколько нестабильны и могут быть выбиты из колеи малейшей ошибкой. Если вы выполнили все вышеизложенные шаги, но все равно при попытке работать с автономным приложением браузер выдает сообщение о невозможности подключения, проверьте, не возникли ли следующие наиболее распространенные проблемы.

- **Проблема с загрузкой манифеста.** Если манифест отсутствует или по каким-либо причинам недоступен для браузера, то это очевидно же будет проблемой. Но также же важно, чтобы для файла манифеста на веб-сервере был установлен правильный MIME-тип *(см. врезку "На профессиональном уровне. Основные сведения о MIME-типах" главы 5)*.
- **Проблемы с загрузкой указанных в манифесте файлов.** Например, манифест может содержать файл изображения, который больше не существует, или файл веб-шрифта, формат которого не разрешается веб-сервером. В любом случае, если браузер не сможет загрузить один-единственный файл, он откажется загружать все остальные файлы и выбросит все файлы, которые уже были кэшированы. Во

избежание этой проблемы начинайте с простых приложений, манифест для которых содержит одну веб-страницу и никаких ресурсов. В более сложных случаях просмотрите журналы веб-сервера, чтобы узнать, какие именно ресурсы были запрошены браузером (что позволит вам определить точку, в которой произошла ошибка, и браузер отказался от продолжения загрузки).

- **Используется старый манифест из кэша.** Браузеры могут сохранить файл манифеста в кэше (согласно обычным правилам кэширования веб-страниц) и игнорировать новый манифест. Одним из признаков этой проблемы будет кэширование одних страниц автономного приложения, но не других, более новых. Эту проблему можно решить, очистив вручную кэш браузера *(см. врезку "Малоизвестная или недооцененная возможность. Очистка кэша браузера" далее в этой главе)*.

Обновление файла манифеста

Заставить автономное приложение работать — это только первая часть задачи. Вторая часть — обновление ее содержимого.

Рассмотрим пример из *разд. "Создание манифеста" ранее в этой главе*, где кэшируются две веб-страницы. Если обновить файл PersonalityTest.html и перезагрузить страницу, браузер все равно будет выводить старую, кэшированную версию страницы, даже если компьютер подключен к Интернету. Проблема состоит в том, что после сохранения автономного приложения в кэше браузер использует только эту кэшированную копию, игнорируя онлайновые версии соответствующих веб-страниц и не проверяя, были ли какие-либо изменения. А так как срок хранения автономного приложения в кэше никогда не истекает, браузер будет упрямо игнорировать измененные страницы на веб-сервере.

Но при этом браузер регулярно проверяет присутствие нового файла манифеста. Таким образом, можно подумать, что все, что нужно сделать для решения этой проблемы, — это сохранить новый файл манифеста на веб-сервере.

Не обязательно. Чтобы инициировать обновление кэшированного автономного приложения, необходимо выполнение следующих требований.

- **Отсутствие кэширования файла манифеста.** Если у браузера есть локальная кэшированная копия файла манифеста, он никогда не будет проверять наличие этого файла на веб-сервере. Разные браузеры подходят по-разному к вопросу кэширования файлов манифеста. Некоторые браузеры (например Chrome) всегда проверяют наличие нового файла манифеста на веб-сервере. Но Firefox следует традиционным правилам кэширования и хранит локальную копию файла манифеста в течение некоторого времени. Поэтому, чтобы избежать проблем в этой области, удостоверьтесь в том, что ваш веб-сервер явно указывает своим клиентам не кэшировать файлы манифеста *(см. разд. "Помещение манифеста на веб-сервер" ранее в этой главе)*.
- **У файла манифеста должна быть проставлена новая дата.** Когда браузер проверяет наличие нового файла манифеста на сервере, первым делом он смотрит, изменилась ли метка времени последнего изменения. Если нет, браузер не будет загружать файл манифеста.

- **Содержимое файла манифеста должно быть новым.** Если браузер обнаружит, что содержимое нового файла манифеста не изменилось, он прекращает процесс обновления и продолжает использовать ранее кэшированную копию. Эта потенциально препятствующая плану обновления практика имеет практическую цель. Повторная загрузка кэшированного приложения занимает время и расходует пропускную способность сети, поэтому браузеры выполняют ее только в тех случаях, когда это абсолютно необходимо.

Если вы внимательно читали этот материал, то должны были заметить здесь потенциальную проблему. Что если менять содержимое файла манифеста нет надобности, т. к. не были добавлены или удалены никакие файлы, но браузер все равно нужно заставить обновить кэш автономного приложения, поскольку содержимое некоторых файлов было изменено? В таких случаях следует незначительно изменить файл манифеста, чтобы он казался новым, хотя, по сути, не является таковым. Эту задачу лучше всего реализовать, вставив в файл манифеста комментарий:

```
CACHE MANIFEST
#version 1.00.001
#pages
PersonalityTest.html
PersonalityTest_Score.html

#styles & scripts
PersonalityTest.css
PersonalityTest.js

#pictures & fonts
Images/emotional_bear.jpg
Fonts/museo_slab_500-webfont.eot
Fonts/museo_slab_500-webfont.woff
Fonts/museo_slab_500-webfont.ttf
Fonts/museo_slab_500-webfont.svg
```

В следующий раз, когда нужно обновить кэш, просто измените номер версии, в этом примере на `1.00.002`, и т. д. Таким образом мы можем заставить браузер обновить кэш и получаем способ отслеживать количество обновлений.

Новое содержимое не отображается в окне браузера сразу же после загрузки. Когда браузер обнаруживает новый файл манифеста, он загружает новые файлы в фоновом режиме и заменяет ими старые кэшированные файлы. Новое содержимое отображается, когда посетитель открывает или обновляет страницу. Чтобы новое содержимое отображалось сразу же после загрузки, можно воспользоваться методом на JavaScript, рассматриваемым в *разд. "Информирование об обновлениях с помощью JavaScript" далее в этой главе*.

Примечание

Обновить автономное приложение по частям невозможно. При малейшем изменении приложения браузер удаляет старые файлы из кэша и заменяет их новыми, включая файлы, которые не были изменены.

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ

Очистка кэша браузера

В процессе тестирования автономного приложения часто бывает полезным вручную очистить кэш браузера. Таким образом можно тестировать свежие обновления без необходимости изменять файл манифеста.

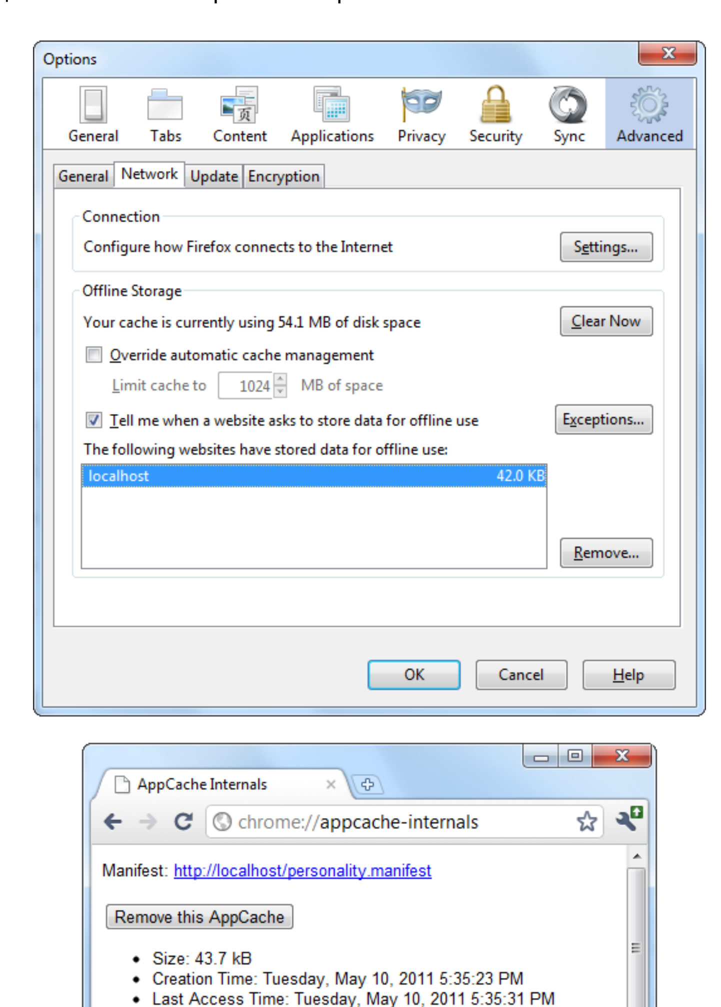


Рис. 10.3. *Вверху:* в браузере Firefox, чтобы просмотреть объем хранилища, занимаемого автономными приложениями, выполните команды меню **Tools | Options**, в открывшемся диалоговом окне **Options** выберите опцию **Advanced** и перейдите на вкладку **Network**. На этой вкладке можно просматривать объем хранилища, занимаемого каждым автономным приложением и при надобности очистить кэш любого из них, выбрав требуемое приложение и нажав кнопку **Remove**. В данном случае имеется только один кэшированный веб-сайт, в домене localhost (что представляет тестовый сервер на локальном компьютере). *Внизу:* для просмотра кэшированных приложений в браузере Chrome введите **about:appcache-internals** в строку адреса

Все браузеры разрешают очистку кэша, но каждый из них подходит к решению этой задачи по-своему. Наиболее полезные в этом отношении браузеры отслеживают объем хранилища, занимаемого каждым автономным приложением (рис. 10.3). Это позволяет определить, когда кэширование было неудачным, например, веб-сайт приложения не указывается в списке или же объем кэшированных файлов меньше, чем он должен быть. Это также позволяет удалять кэшированные файлы для отдельных сайтов, не затрагивая другие.

Браузерная поддержка автономных приложений

К этому времени вы, наверное, уже поняли, что автономные приложения поддерживаются всеми основными браузерами, за обычным исключением — Internet Explorer. Поддержка распространяется на несколько версий назад, что практически обеспечивает возможность пользователей браузеров Firefox, Chrome и Safari исполнять автономные приложения. Минимальные версии основных браузеров, поддерживающих автономные приложения, перечислены в табл. 10.1.

Таблица 10.1. *Браузерная поддержка автономных приложений*

	IE	**Firefox**	**Chrome**	**Safari**	**Opera**	**Safari iOS**	**Android**
Минимальная версия	—	3.5	5	4	10.6	2.1	2

Но различные браузеры поддерживают автономные приложения по-разному. Самая большая разница состоит в объеме хранилища, выделяемого автономным приложениям для кэширования своих файлов. Это важный аспект, т. к. он определяет, какие веб-сайты будут помещены в кэш для автономного доступа, а какие — нет *(см. врезку "Часто задаваемый вопрос. Каким может быть объем кэша?" ранее в этой главе)*.

Если браузер не имеет встроенной поддержки автономных приложений (например, Internet Explorer), предоставить такую поддержку каким-либо обходным решением невозможно. Но это обстоятельство не должно останавливать вас от использования возможности работы ваших веб-сайтов в автономном режиме. В конце концов режим автономной работы — хоть и полезная, но всего лишь необязательная возможность.

Браузеры, не поддерживающие автономных приложений, все равно будут работать с такими сайтами, только будут требовать для этого подключения к Интернету. А пользователи, нуждающиеся в возможности автономной работы с сайтом, например, те, кто часто путешествуют, откроют для себя, насколько это ценно пользоваться браузером иным, чем Internet Explorer.

Практические методы кэширования

Итак, мы узнали, как упаковать группу страниц и ресурсов в виде автономного приложения. Мы научились создавать файл манифеста, обновлять его и обеспечи-

вать исполнение созданного автономного приложения в браузерах. Хотя этих знаний достаточно для создания простых автономных приложений, более сложные веб-сайты иногда требуют дополнительной работы. Например, мы можем хранить определенное содержимое онлайн, заменять определенные страницы в автономном режиме или определять (программным способом), подключен ли компьютер к Интернету. Далее мы научимся реализовывать все эти задачи посредством более продвинутых манифестов и применением кода JavaScript.

Доступ к онлайновым файлам

Мы узнали, что после кэширования страницы браузер использует эту копию страницы и не обращается к серверу, чтобы узнать, нет ли ее обновленной версии. Вы, возможно, не подозреваете, что браузер отказывается обращаться к серверу за всеми используемыми автономным приложением ресурсами независимо от того, кэшированы они или нет.

Допустим, например, что на нашей автономной странице используются два изображения:

```
<img src="Images/logo.png" alt="Personality quiz">
<img src="Images/emotional_bear.jpg" alt="Sad stuffed bear">
```

Но манифест кэширует только одно из этих изображений:

```
CACHE MANIFEST
PersonalityTest.html
PersonalityTest_Score.html

PersonalityTest.css PersonalityTest.js
Images/emotional_bear.jpg
```

Размышляя логически, можно предположить, что браузер возьмет изображение emotional_bear.png из своего кэша, а изображение logo.png запросит у веб-сервера (при условии, что компьютер подключен к Интернету). Ведь так происходит при работе с обычными сайтами, когда браузер обращается с кэшированной страницы к странице, которая еще не была сохранена в кэше. Но в действительности в этом отношении автономные приложения работают по-другому. Браузер берет изображение emotional_bear.jpg из кэша, но игнорирует графику logo.png, отображая вместо нее либо специальный значок отсутствующего изображения, либо просто пустое место, в зависимости от браузера.

Эта проблема решается добавлением в манифест нового раздела. Он начинается с заголовка `NETWORK:`, а за ним следует список файлов, которые нужно получать с веб-сервера:

```
CACHE MANIFEST
PersonalityTest.html
PersonalityTest_Score.html

PersonalityTest.css
PersonalityTest.js
```

```
Images/emotional_bear.jpg

NETWORK:
Images/logo.png
```

Теперь, когда компьютер подключен к Интернету, браузер будет пытаться получить файл logo.png с веб-сервера, но при отсутствии подключения предпринимать такую попытку не будет.

Здесь у вас, наверное, возникает вопрос: зачем утруждать себя созданием явного списка файлов, которые мы не хотим кэшировать? Одна из причин — из-за возможностей хранилища, например, мы игнорируем файлы большого объема, чтобы наше приложение могло сохраниться в кэше браузеров, разрешающих максимум 5 Мбайт кэша для каждого автономного приложения.

Но более вероятной будет ситуация, когда у нас есть содержимое, которое должно предоставляться, когда запрашивается, но никогда не должно кэшироваться, например, отслеживающие сценарии или динамически создаваемая реклама. В таком случае, самым легким решением будет добавить звездочку (`*`) в раздел `NETWORK`. Это знак подстановки, указывающий браузеру обращаться к серверу за всеми ресурсами, которые не были явно кэшированы:

```
NETWORK:
*
```

Знак подстановки можно также использовать, чтобы указывать файлы конкретного типа (например, выражение `*.jpg` обозначает все изображения JPEG) или все файлы на определенном сервере (например, выражение `http://www.google-analytics.com/*` обозначает все ресурсы в домене Google Analytics).

Примечание

У вас может возникнуть мысль, что манифест можно было бы упростить, используя знак подстановки для указания всех файлов, которые нужно сохранить в кэше, вместо того, чтобы задавать каждый отдельный файл. К сожалению (или к счастью), указание файлов для кэширования посредством звездочки не поддерживается, т. к. создатели HTML5 беспокоились, что небрежные веб-разработчики могут пытаться поместить в кэш огромные веб-сайты.

Добавление резервных решений

Манифест указывает браузерам, какие файлы нужно кэшировать, а какие получать с сервера (перечисляются в разделе `NETWORK`). Манифест также может содержать раздел для указания резервных решений, т. е. замены одного файла другим, в зависимости от наличия или отсутствия подключения к Интернету.

Раздел резервных решений начинается с заголовка `FALLBACK:`, который можно расположить в любом месте манифеста. После заголовка дается попарный список файлов. Первый файл пары используется при наличии подключения к Интернету, а второй является резервным и выбирается при отсутствии подключения:

```
FALLBACK:
PersonalityScore.html PersonalityScore_offline.html
```

Браузер загружает резервный файл (в данном случае это файл PersonalityScore_offline.html) и кэширует его. Но браузер использует этот файл только в том случае, если компьютер не подключен к Интернету, а при наличии подключения браузер запрашивает первый файл (т. е. в данном случае PersonalityScore.html).

> ***Примечание***
>
> Не забывайте, что компьютер не обязательно должен быть отключен от Интернета, чтобы автономное приложение находилось в режиме оффлайн. В этом отношении важной является доступность данного домена. Если домен не отвечает по любым причинам, автономное приложение считает себя оффлайновым.

Причин для использования резервных файлов большое множество. Например, для автономной работы основную страницу можно заменить более простой страницей, которая не использует сложные сценарии или ресурсы большого объема. Раздел резервных файлов можно разместить в любом месте манифеста при условии, что он обозначается соответствующим заголовком `FALLBACK:`.

```
CACHE MANIFEST
PersonalityTest.html
PersonalityTest_Score. html

PersonalityTest.css

FALLBACK:
PersonalityScore.html PersonalityScore_offline.html
Images/emotional_bear.jpg Images/emotional_bear_small.jpg
PersonalityTest.js PersonalityTest_offline.js

NETWORK:
*
```

> ***Примечание***
>
> Кстати, раздел манифеста, в котором перечисляются файлы для кэширования, называется `CACHE:`. При желании этот заголовок можно указывать, но это не обязательно, если только список файлов для кэширования не задается после одного из других разделов.

Раздел резервных файлов также поддерживает использование подстановочного символа. Эта возможность позволяет создавать встроенную страницу с сообщением об ошибках:

```
FALLBACK:
/ offline.html
```

Теперь допустим, что браузер запрашивает страницу, находящуюся на том же веб-сайте, что и автономное приложение, но которая не была кэширована. Если компьютер подключен к Интернету, браузер попытается обратиться к серверу и получить

живую страницу. Но при отсутствии подключения, или если веб-сайт недоступен по какой-либо другой причине, или запрашиваемой страницы просто нет на сервере, браузер отображает кэшированную страницу offline.html (рис. 10.4).

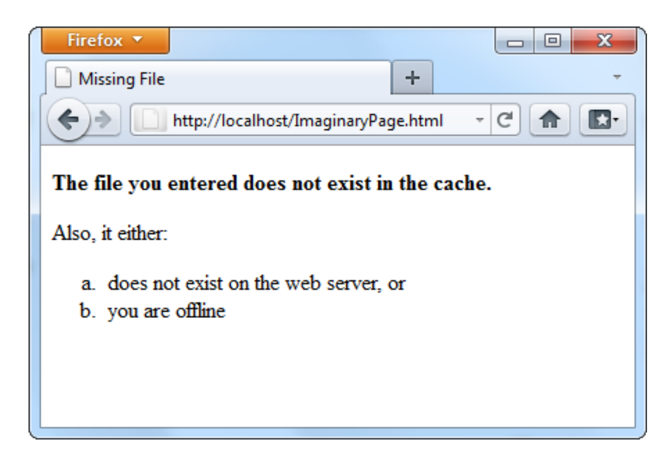


Рис. 10.4. В данном примере страница ImaginaryPage.html не существует. Интересно, что браузер не обновляет соответствующим образом информацию в строке адреса, вследствие чего посетитель не знает настоящее название отображаемой страницы с сообщением об ошибке доступа

В только что рассмотренном примере для обозначения любой страницы применяется несколько произвольная нотация подстановочного символа в виде косой черты (/). Это может показаться несколько странным в виду того, что в разделе `NETWORK:` для этой же цели применяется звездочка. В действительности некоторые браузеры (например, Firefox) разрешают использование звездочки в качестве подстановочного символа в разделе резервных файлов. Это означает, что для таких браузеров предыдущий пример можно переделать:

```
FALLBACK:
* offline.html
```

Кроме этого, можно указать резервные ресурсы, совпадающие со всеми файлами в указанной папке:

```
FALLBACK:
http://www.superAppsOnSteroids.org/paint_app/* offline.html
```

Или же ресурсы, совпадающие с определенными типами файлов:

```
FALLBACK:
*.jpg missing_picture.jpg
```

К сожалению, другие браузеры не понимают этот логичный синтаксис, по крайней мере, пока еще не понимают.

Проверка подключения

Раздел указания резервных файлов можно использовать в удобном JavaScript-методе для определения, находится ли браузер в режиме онлайн или нет. Опытные

разработчики JavaScript, наверное, помнят свойство `navigator.onLine`, предоставляющее слегка ненадежный способ проверки, находится ли браузер в настоящее время в онлайн. Но проблема с этим методом состоит в том, что свойство `onLine` в действительности отображает состояние параметра браузера "Работать автономно", а не собственно наличие или отсутствие подключения к Интернету. Но даже если бы свойство `onLine` было более надежным индикатором наличия подключения, оно все равно не могло бы сообщить нам, браузер не загрузил страницу потому, что он не смог подключиться к серверу, или по какой-либо другой причине.

Эта проблема решается с помощью резервного решения, которое загружает разные версии одной и той же функции JavaScript в зависимости от того, находится ли приложение онлайн или оффлайн. Раздел `FALLBACK:` манифеста будет таким:

```
FALLBACK:
online.js off line.js
```

Первоначальная версия веб-страницы ссылается на JavaScript-файл online.js:

```
<!DOCTYPE html>
<html lang="en" manifest="personality.manifest">
<head>
  <meta charset="utf-8">
  <title>...</title>
  <script src="online.js"></script>
  ...
```

Она содержит эту очень простую функцию:

```
function isSiteOnline() {
  return true;
}
```

Но если файл online.js недоступен, браузер использует вместо него файл offline.js, который содержит ту же функцию, но с другим результатом:

```
function isSiteOnline() {
  return false;
}
```

В исходной странице, когда нужно узнать, находится ли приложение в режиме онлайн, выполняем эту проверку с помощью функции `isSiteOnline()`:

```
var displayStatus = document.getElementById("displayStatus");
if (isSiteOnline()) {
  // (Можно выполнять задачи, требующие подключения к Интернету, такие
  // как взаимодействие с сервером с помощью объекта XMLHttpRequest.)
  displayStatus.innerHTML = "Вы подключены к Интернету, " +
                            "и веб-сервер доступен.";
}
else {
  // (Приложение исполняется в автономном режиме. Может быть,
  // следует скрыть или программным способом изменить определенное
  // содержимое или отключить некоторые возможности.)
```

```
  displayStatus.innerHTML = "Это приложение исполняется " +
                            "в автономном режиме.";
}
```

Информирование об обновлениях с помощью JavaScript

С автономным приложением можно взаимодействовать посредством сравнительно ограниченного интерфейса на основе JavaScript-объекта `applicationCache`.

Свойство `status` объекта `applicationCache` указывает, какую операцию браузер выполняет в настоящее время — проверяет наличие обновленного манифеста, загружает новые файлы и т. п. Значение этого свойства часто обновляется, и оно почти такое же полезное, как и взаимодействующие события (табл. 10.2), которые срабатывают при изменении статуса объекта `applicationCache`.

Таблица 10.2. *События кэширования*

Событие	Описание
`onChecking`	Когда браузер обнаруживает в веб-странице атрибут `manifest`, он активирует это событие и проверяет наличие соответствующего файла манифеста на веб-сервере
`onNoUpdate`	Если браузер уже загрузил файл манифеста и этот файл не был изменен, он активирует это событие и не предпринимает никаких дальнейших действий
`onDownloading`	Браузер активирует это событие, прежде чем начинать загрузку манифеста (и указанных в манифесте файлов). Это происходит при первой загрузке файлов манифеста и при обновлениях
`onProgress`	Браузер периодически активирует это событие при загрузке файлов, чтобы информировать о ходе загрузки
`onCached`	Событие указывает на завершение первоначальной загрузки для нового автономного приложения. После этого больше не происходит никаких событий
`onUpdateReady`	Событие указывает на завершение загрузки обновленного содержимого. На этом этапе новое содержимое готово к использованию, но не отображается в окне браузера до тех пор, пока страница не будет перезагружена. После этого больше не происходит никаких событий
`onError`	Где-то произошла ошибка. Возможно, недоступен веб-сервер (в таком случае страница переключается в автономный режим), возможно, в манифесте использован неправильный синтаксис или недоступен кэшированный ресурс. После этого события не происходит больше никаких других событий
`onObsolete`	В процессе проверки на наличие обновлений браузер обнаружил отсутствие манифеста, после чего он очищает кэш. При следующей загрузке страницы браузер загрузит ее с веб-сервера

Примечание

На момент написания этих строк различные браузеры предоставляли разный уровень поддержки событий кэширования. Например, браузер Firefox игнорирует такие полезные события, как `onChecking` и `onUpdateReady`, но активирует события `onNoUpdate` и `onError`.

Наиболее полезным событием является событие `onUpdateReady`, которое сигнализирует о загрузке новой версии автономного приложения. Хотя новая версия готова к использованию, в окне браузера все еще отображается старая версия. В таком случае желательно дать пользователю знать, что новая версия приложения готова к использованию. Для этого можно применить следующий код:

```
<script>
  window.onload = function() {

  // Подключаем обработчик события onUpdateReady.
  applicationCache.onupdateready = function()  {
    var displayStatus = document.getElementById("displayStatus");

    // Доступна новая версия приложения.
    // Чтобы загрузить ее, обновите страницу.
    displayStatus.innerHTML = "There is a new version of " +
                              "this application. To load it, " +
                              "refresh the page.";
  }
  }
</script>
```

Или же можно предложить обновить страницу программным способом, используя метод `window.location.reload()`:

```
<script>
  window.onload = function() {

  applicationCache.onupdateready = function() {
    if (confirm(
      // Доступна новая версия приложения. Загрузить ее сейчас?
      "A new version of this application is available. Reload now?")){
        window.location.reload();
      }
  }
  }
</script>
```

Результаты выполнения последнего примера показаны на рис. 10.5.

Объект `applicationCache` также предоставляет два метода для более специализированных сценариев. Первый метод, `update()`, проверяет наличие нового манифеста. В случае его существования метод начинает загрузку в фоновом режиме. Иначе не предпринимается никаких дальнейших действий.

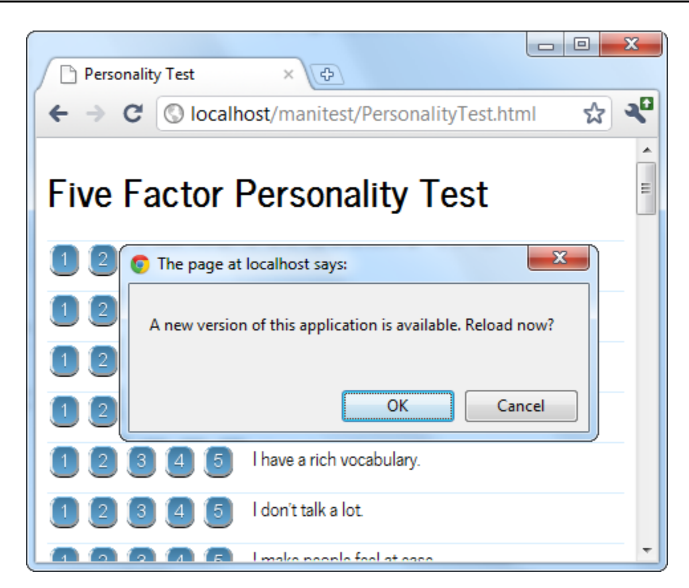


Рис. 10.5. Приложение предлагает пользователю отобразить обновленное содержимое, нажав кнопку **OK**. В противном случае новое содержимое будет выведено в окно при следующей загрузке страницы или ее обновлении

Хотя браузеры проверяют наличие обновлений автоматически, если есть основания полагать, что манифест изменился после первой загрузки страницы, метод `update()` можно вызвать вручную. Этот метод может быть полезным в приложениях с продолжительным временем работы, например, когда пользователь работает на одной и той же странице в течение всего дня.

Второй метод, `swapCache()`, указывает браузеру начать использование нового содержимого кэша после загрузки обновления. Но этот метод не обновляет отображаемое содержимое страницы, для этого нужно перезагрузить или обновить страницу. Какая же тогда от этого метода польза? Переключение на новое содержимое кэша обеспечивает выборку всего загружаемого с этого момента содержимого, например динамически загружаемого изображения, из нового кэша, а не из старого. При аккуратном использовании метод `swapCache()` может позволить странице загружать новое содержимое, не требуя полной перезагрузки страницы (в процессе, возможно, сбрасывая все настройки приложения в исходное состояние). Но в большинстве приложений метод `swapCache()` доставляет больше хлопот, чем пользы и может вызывать неочевидные ошибки, смешивая новые и старые части кэша.

11

ГЛАВА 11

Взаимодействие с веб-сервером

Мы начали изучать HTML5 с возможностей маркировки, таких как семантические элементы, веб-формы и видео. Но по мере овладения материалом, мы постепенно смещали наше внимание на программирование веб-страниц и на части HTML5, движимые JavaScript. А теперь мы затронем новые возможности HTML5, которые выводят веб-программирование на новый уровень. Для этих возможностей требуется не только применение JavaScript, но и немного помощи со стороны серверных программ. Серверная программа — это программа, которая исполняется на сервере и может быть на любом из языков серверного программирования.

Добавление в наш инструментарий серверного программирования представляет небольшую проблему. С одной стороны, выбор языка серверного программирования не имеет значения при условии, что он может работать с чисто HTML5-страницами (а все эти языки могут). Но с другой стороны, бессмысленно влезать по уши в изучение новой технологии, которую вы не планируете использовать или которая не поддерживается вашим веб-хостом. А хороших языков серверного программирования предостаточно, включая PHP, ASP.NET, Ruby, Java, Python и многие другие.

В данной главе эта проблема решается посредством использования очень небольшого объема весьма простого кода серверного программирования. Этого кода как раз достаточно, чтобы завершить каждый пример и позволить протестировать HTML5-сторону. Вы сможете модифицировать и расширить этот код на своих веб-сайтах в зависимости от ваших целей и предпочитаемого языка серверного программирования.

Так что же это за возможности, требующие вмешательства со стороны сервера? Спецификация HTML5 регламентирует два новых способа взаимодействия веб-страниц с сервером. Первым из них являются *отправляемые сервером события* (server-sent events), посредством которых сервер может периодически связываться с веб-страницей и передавать ей информацию. Второй способ, более амбициозный, называется инфраструктурой *веб-сокетов* (web sockets) и позволяет свободный двусторонний обмен информацией между сервером и браузером. Но прежде чем приступить к изучению этих возможностей, мы вкратце рассмотрим средство, при-

меняемое в настоящее время для реализации взаимодействия между веб-серверами и их клиентами: объект `XMLHttpRequest`.

Примечание

Технологии отправляемых сервером событий и веб-сокетов выглядят обманчиво простыми. Разобраться, как они работают, и создать очень простой пример (как в этой главе) достаточно легко. Но расширить эти начальные знания и приобрести опыт, чтобы создать решение, которое будет надежно работать на профессиональном сайте и предоставлять требуемые вам функциональности, совсем нелегко. Из всего этого можно сделать следующий вывод: чтобы реализовать эти возможности на своем веб-сайте, вам, вероятно, придется обратиться за помощью к кому-то, кто обладает солидным опытом в области серверного программирования.

Отправка сообщений на веб-сервер

Прежде чем мы сможем понять новые возможности HTML5 в области взаимодействия с сервером, нам нужно знать, каковой была ситуация в этой области раньше. Это означает изучение важного JavaScript-объекта `XMLHttpRequest`, посредством которого веб-страницы могут взаимодействовать с веб-сервером. Если вы уже знакомы с этим объектом (и используете его в своих страницах), то можете пропустить материал данного раздела. Но если ваша карьера веб-разработчика пока состоит из разработки только более традиционных решений, вам нужно ознакомиться с этим материалом, чтобы получить необходимые базовые знания.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Развитие взаимодействия с веб-сервером

На ранних этапах становления Интернета взаимодействие с веб-сервером было простым делом: браузер просто запрашивал веб-страницу, а веб-сервер отправлял ему эту страницу. Вот и все взаимодействие.

Но немного позже разработчики программного обеспечения начали проявлять изобретательность. Они разработали серверные средства, которые можно было помещать между первым шагом (запрос страницы браузером) и вторым шагом (отправка страницы браузером), исполняя определенный код на веб-сервере. Идея заключалась в том, чтобы динамически изменять страницу (например, удаляя из нее блок разметки) или даже создавать полностью новую страницу (например, считывая запись из базы данных и генерируя специальную страницу, содержащую информацию о продукте).

Потом веб-программисты вошли во вкус и захотели создавать страницы, обладающие еще большей интерактивностью. Эту задачу можно было реализовать с помощью инструментов серверного программирования (немного похимичив) при условии, что браузер не возражал против обновления страницы. Например, если посетитель интернет-магазина поместил новый товар в свою корзину, это событие нужно отобразить. Для этого информация о новом товаре в корзине отправляется на веб-сервер (посредством веб-форм, рассматриваемых в *главе 4*), а веб-сервер отправляет браузеру обновленную страницу, отображающую текущее содержимое корзины покупок. Эта стратегия получила большой успех, однако была несколько неуклюжей.

Но веб-разработчики на этом не остановились и продолжали изыскивать новые способы улучшения интерактивности между браузером и веб-сервером. Теперь они хотели иметь в своем распоряжении средство для создания веб-приложений (наподобие программ электронной почты), в которых не нужно было бы постоянно обновлять пол-

ностью всю страницу. Решение было найдено в виде набора методов, которые иногда называются AJAX, но почти всегда связаны со специальным JavaScript-объектом `XMLHttpRequest`. Посредством этого объекта веб-страницы могут обратиться к серверу, отправить ему данные и получить от него ответ, не обновляя не то что всей страницы, но вообще никакой ее части. Это расчищает дорогу для управления посредством JavaScript всеми аспектами страницы, включая обновление ее содержимого. Эта технология также позволяет сделать веб-страницы более элегантными и оперативными.

Объект *XMLHttpRequest*

Основным инструментом, который предоставляет веб-страницам возможность взаимодействовать с сервером, является объект `XMLHttpRequest`. Объект `XMLHttpRequest` изначально был создан корпорацией Microsoft для того, чтобы улучшить веб-версию своей программы электронной почты Outlook, но он постепенно распространился на все современные браузеры. В настоящее время он является основной частью большинства современных веб-приложений.

Ключевая идея в основе объекта `XMLHttpRequest` состоит в том, что он позволяет коду JavaScript самостоятельно направлять запросы к серверу, когда приложению требуются дополнительные данные. Эти запросы осуществляются асинхронно, что означает, что веб-страница остается доступной для работы даже в процессе выполнения такого запроса. Более того, посетитель страницы никогда даже не догадывается о выполняющемся за кулисами запросе (если только не выводится соответствующее извещение или индикатор хода выполнения).

Объект `XMLHttpRequest` является идеальным инструментом для получения данных с веб-сервера. Далее приведено несколько примеров данных, которые можно получить с веб-сервера посредством этого объекта.

- **Данные, хранящиеся на сервере.** Это может быть информация в файле или, в большинстве случаев, в базе данных. Например, посетитель может затребовать информацию о продукте или о компании.
- **Данные, которые можно вычислить только на сервере.** Например, веб-сайт может содержать на сервере программу, выполняющую сложные вычисления. Эти вычисления можно было бы попытаться выполнить в браузере с помощью JavaScript, но по разным причинам это может оказаться невозможным. Например, математические возможности JavaScript могут быть ниже требуемого уровня или же код может не иметь доступа к некоторым данным, требующимся для вычислений. Также код может выполнять секретные вычисления, которые вы не хотите выставлять напоказ для всех, кто пожелает их видеть, чтобы избежать потенциальных несанкционированных манипуляций с вычислениями. Или же вычисления могут требовать возможностей, которые настольный компьютер не в силах предоставить, как это может сделать мощный веб-сервер (например, прорисовка трехмерной сцены). Во всех подобных случаях имеет смысл выполнять вычисления на сервере.
- **Данные, которые находятся на чьем-то другом сервере.** Веб-страница не может обращаться напрямую к чьему-то другому веб-серверу. Но она может вы-

звать программу на своем веб-сервере (посредством объекта XMLHttpRequest), которая в свою очередь может вызвать другой веб-сервер, получить от него данные и возвратить эти данные на исходную страницу.

Лучший способ разобраться с объектом XMLHttpRequest — это начать экспериментировать с ним. Этим мы и займемся в последующих разделах, рассмотрев два простых примера.

Отправка запроса веб-серверу

На рис. 11.1 показана веб-страница, которая просит веб-сервер выполнить простое математическое вычисление. Этот запрос отправляется веб-серверу посредством объекта XMLHttpRequest.

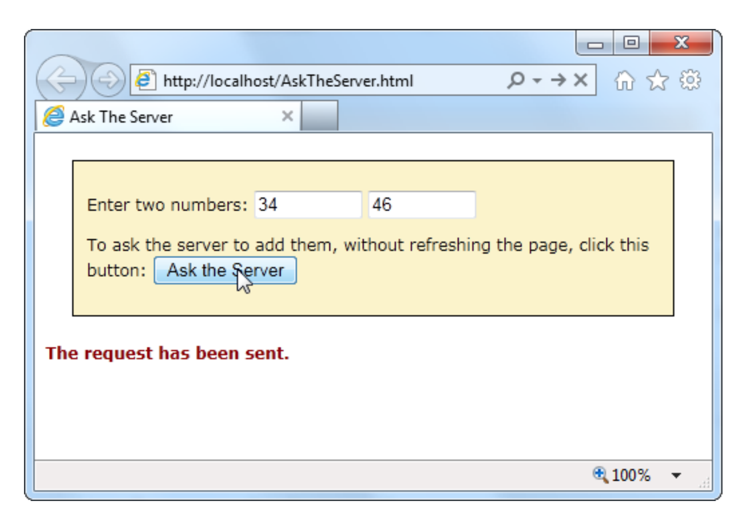


Рис. 11.1. По нажатию кнопки **Ask the Server** веб-страница создает объект XMLHttpRequest и отправляет два числа на веб-сервер. Веб-сервер отрабатывает простой сценарий, выполняющий математическую операцию над числами, и отправляет ответ обратно на веб-страницу (см. рис. 11.2)

Прежде чем мы сможем создать эту страницу, нам понадобится какой-либо серверный сценарий для обработки полученной от веб-страницы информации (в данном случае два введенных в нее числа) и отправки на нее полученного результата вычислений. Эту задачу можно выполнить на любом существующем языке серверного программирования (в конце концов, отправить небольшой фрагмент текста намного легче, чем целый документ HTML). В нашем примере используется сценарий PHP, в большей мере потому, что это сравнительно простой язык, который поддерживается почти всеми хостинговыми компаниями.

Создание сценария

Сценарий PHP создается в простом текстовом файле и имеет следующую структуру:

```
<?php
  // (Код вставляется сюда.)
?>
```

Для данного примера код для выполнения вычисления и предоставления результатов донельзя простой и выглядит так:

```
<?php
  $num1 = $_GET['number1'];
  $num2 = $_GET['number2'];
  $sum = $num1 + $num2
  echo($sum);
?>
```

Даже если вы и не эксперт по PHP, у вас, скорее всего, не будет трудностей разобраться, что этот код делает. Первым делом мы получаем два числа, отправленные веб-страницей:

```
$num1 = $_GET['number1'];
$num2 = $_GET['number2'];
```

Знак доллара (`$`) обозначает переменную, поэтому код создает две переменные: `$num1` и `$num2`. Значения для переменных код извлекает из встроенной в PHP коллекции, называющейся `$_GET`. Эта коллекция содержит всю информацию из URL, посредством которой был запрошен данный сценарий.

Например, если поместить сценарий PHP в файл WebCalculator.php, строка URL с запросом этого сценария будет выглядеть так:

```
http://www.magicalXMLHttpRequestTest.com/
       WebCalculator.php?number1=34&number2=46
```

В данном случае URL содержит две единицы информации в конце (в части URL, называющейся *строкой запроса*, — query string). Значение первой переменной в этой части, `number1`, равно `34`, а второй, `number2`, равно `46`. Начало строки запроса обозначается знаком вопроса (`?`), а каждая последующая переменная — знаком амперсанда (`&`). Когда код PHP начинает исполняться, он извлекает эту информацию из URL и сохраняет ее в коллекции `$_GET`, где она доступна для последующих операций. (Большинство платформ серверного программирования поддерживает модель, подобную этой. Например, в технологии ASP эта информация сохраняется в коллекции `Request.QueryString`.)

Примечание

Ветераны HTML знают, что данные на веб-сервер можно отправлять двумя способами — посредством строки запроса или вставкой их в тело запроса. В любом случае данные кодируются одинаково, и доступ к ним на сервере предоставляется похожим образом. Например, в PHP данные, переданные в теле запроса, помещаются в коллекцию `$_POST`.

Получив эти два числа, сценарий PHP просто суммирует их:

```
$sum = $num1 + $num2
```

Последний шаг — отправить результаты назад веб-странице, которая подала запрос. Эти результаты можно было бы обернуть в разметку HTML или даже в приспособленную для данных разметку XML, но это будет слишком для данного приме-

ра, и простого текста будет вполне достаточно. Но независимо от выбранного формата данных, все, что требуется для их отправки, — это простая PHP-команда `echo`:

```
echo($sum);
```

Итак, сценарий состоит из всего четырех строк PHP-кода. Но этого достаточно, чтобы установить основной шаблон: веб-страница задает веб-серверу вопрос, а веб-сервер предоставляет ответ на этот вопрос.

Примечание

Простота вычислений в данном примере может породить вопрос: нельзя ли выполнить их полностью в браузере посредством JavaScript, а не прибегать к помощи сервера? Конечно же, можно. Но в данном случае важно не собственно вычисление. Рассмотренный сценарий PHP служит примером для выполнения любой задачи, которую нужно выполнить на сервере. Вычисления могут быть любой сложности, но базовый образец обмена данными будет тем же.

Обращение к веб-серверу

Вторым шагом будет создание страницы, которая с помощью объекта `XMLHttpRequest` использует сценарий PHP. Код начинается достаточно просто. Сначала создается объект `XMLHttpRequest` для использования во всех функциях:

```
var req = new XMLHttpRequest();
```

При нажатии кнопки **Ask the Server** вызывается функция `askServer()`:

```
<div> <p>Enter two numbers:
         <input id="number1" type="number">
         <input id="number2" type="number">
      </p>
  <p>To ask the server to add them, without refreshing the page,
     click this button:<button onclick="askServer()">
     Ask the Server</button>
  </p>
</div>
<p id="result"></p>
```

Посредством объекта `XMLHttpRequest` функция `askServer()` выполняет запрос в фоновом режиме. Сначала она получает требуемые данные — два числа:

```
function askServer() {
  var number1 = document.getElementById("number1").value;
  var number2 = document.getElementById("number2").value;
```

А потом создает из этих данных строку запроса:

```
  var dataToSend = "?number1=" + number1 + "&number2=" + number2;
```

Теперь все готово для создания запроса, который начинается методом `open()` объекта `XMLHttpRequest`. В качестве параметров методу передается тип операции HTTP (`GET` или `POST`), URL запроса и значение `true` или `false`, определяющее режим выполнения запроса браузером — асинхронно или нет.

```
req.open("GET", "WebCalculator.php" + dataToSend, true);
```

ПРИМЕЧАНИЕ

Веб-эксперты сходятся в едином мнении — последний параметр метода `open()` всегда должен быть `true`, что означает асинхронное выполнение запроса. Причиной этому то обстоятельство, что ни один веб-сайт не является полностью надежным, и синхронный запрос (т. е. запрос, который заставляет код ожидать ответа) может потенциально вызвать сбой всей страницы, пока он ожидает ответа.

Но прежде чем отправлять запрос, к событию `onReadyStateChange` объекта `XMLHttpRequest` необходимо подключить функцию. Это событие активируется при получении информации от сервера, включая конечный результат вычислений:

```
req.onreadystatechange = handleServerResponse;
```

Теперь можно начинать процесс, вызвав метод `send()` объекта `XMLHttpRequest`. Но помните, что код продолжает исполняться без каких-либо задержек. Единственный способ получить ответ — это обратиться к событию `onReadyStateChange`, которое можно будет активировать позже:

```
  req.send();

  document.getElementById("result").innerHTML=
    "The request has been sent.";
}
```

Получив ответ, нужно немедленно проверить два свойства объекта `XMLHttpRequest`. Сначала проверяется свойство `readyState`, значение которого изменяется от 0 до 4 в процессе выполнения запроса. При инициализации запроса это значение равно 1, при отправке — 2, при частичном получении ответа — 3, при завершении выполнения — 4. Если значение свойства `readyState` не равно 4, нет смысла продолжать дальнейшую обработку и нужно проверить значение свойства `status`, которое предоставляет код состояния HTTP. Для наших целей значение этого свойства должно быть 200, что означает отсутствие каких-либо проблем. Значение этого свойства будет равно 401 при запросе неразрешенной страницы; 404, если запрошенная страница не была найдена; 302, если страница была перемещена; или 503, если страница занята. (Полный список кодов ошибок HTTP см. по адресу **www.addedbytes.com/for-beginners/http-status-codes**.)

Код для проверки значения этих свойств выглядит так:

```
function handleServerResponse() {
  if ((req.readyState == 4) && (req.status == 200)) {
```

Если свойства содержат требуемые значения, можно извлечь результаты из свойства `responseText` объекта `XMLHttpRequest`. В нашем случае это свойство содержит сумму двух исходных чисел. Полученный результат отображается на странице (рис. 11.2) следующим кодом:

```
    var result = req.responseText;
    document.getElementBy!d("result").innerHTML =
                          "The answer is " + result + ".";
  }
}
```

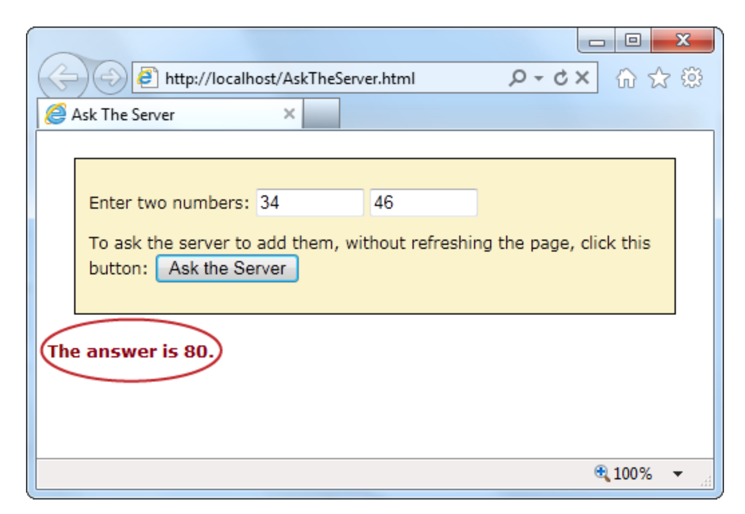


Рис. 11.2. Веб-сервер возвратил ответ, активировав функцию JavaScript, которая отобразила полученный ответ на странице

Объект `XMLHttpRequest` не предполагает какого-либо конкретного типа запрашиваемых данных. Название объекта содержит часть XML по той причине, что он был создан для работы с данными XML, т. к. этот стандарт позволяет создавать удобные логические пакеты структурированной информации. Но объект `XMLHttpRequest` также используется для запросов простого текста (как в только что рассмотренном примере), JSON-данных *(см. разд. "Сохранение объектов" главы 9)*, данных HTML (как в следующем примере) и данных XML. В действительности, в настоящее время объект `XMLHttpRequest` используется намного чаще для работы с данными, отличными от XML, так что пусть его название не сбивает вас с толку.

Совет

Для функционирования страниц, содержащих серверный код, включая сценарии PHP, они должны быть размещены на веб-сервере. Чтобы избежать проблем с испытанием примеров в этой главе, запускайте их на сайте этой книги **http://www.prosetech.com/html5/**.

Получение нового содержимого

Другой ситуацией, в которой можно использовать объект `XMLHttpRequest`, будет загрузка нового содержимого в страницу. Например, новостная статья может сопровождаться несколькими фотографиями, но одновременно отображается только одна из них. Чтобы отобразить другую фотографию, пользователь нажимает соответствующую кнопку, а JavaScript-код получает ее и вставляет вместо предыдущей. Этот же способ можно использовать для показа изображений в слайд-шоу.

Пример такого слайд-шоу показан на рис. 11.3.

Для использования дизайна, подобного показанному на рис. 11.3, может быть несколько причин. При умелой реализации этот метод может позволить удерживать под контролем большие объемы содержимого, которое всегда готово для просмот-





Рис. 11.3. Содержимое этой страницы состоит из нескольких отдельных слайдов. Управление слайдами осуществляется щелчком по ссылке **Next** или **Previous**, в результате чего загружается новое изображение со своей подписью. Запрос нового содержимого осуществляется посредством объекта `XMLHttpRequest`

ра, но не перегружает страницу. (Менее умелые руки могут применить этот метод для накрутки количества посещений своей страницы, заставляя посетителя выполнять несколько запросов, чтобы получить все требуемое содержимое.)

Лучшим способом реализации страницы такого типа будет использование объекта `XMLHttpRequest`. Таким образом, страница может запрашивать новое содержимое и обновлять только часть окна, не вызывая обновления всей страницы. А полные обновления плохи тем, что расходуют трафик, вызывают мигание страницы и заставляют пользователя прокручивать страницу обратно вверх. Все это может казаться мелочами, но такие мелочи отличают профессиональные, отполированные страницы от неуклюжих любительских.

Для создания примера, показанного на рис. 11.3, сначала нам нужно выделить место для динамического содержимого. Для этого мы используем элемент `<div>`, который создает нам рамку с золотистым обрамлением и двумя ссылками внизу:

```
<div id="slide">Click Next to start the show.</div>
<a onclick="return previousSlide()" href="#">&lt; Previous</a> 
<a onclick="return nextSlide()" href="#">Next &gt;</a>
```

Ссылки вызывают функцию `previousSlide()` или `nextSlide()` в зависимости от направления просмотра слайдов. В обеих функциях используется счетчик, начальное значение которого равно `0`, увеличивается до `5` и начинает новый цикл со значения `1`. Код функции `nextSlide()` выглядит следующим образом:

```
var slideNumber = 0;

function nextSlide() {
  // Увеличиваем указатель слайдов.
  if (slideNumber == 5)  {
    slideNumber = 1;
  } else {
    slideNumber += 1;
  }

  // Вызываем функцию для показа слайда.
  goToNewSlide();

  // Гарантируем, что ссылка не выполняет никаких действий.
  return false;
}
```

Код функции `previousSlide()` подобен функции для показа следующего слайда:

```
function previousSlide() {
  if (slideNumber == 1) {
    slideNumber = 5;
  } else {
    slideNumber -= 1;
  }
```

```
  goToNewSlide();
  return false;
}
```

Обе функции используют еще одну функцию, `goToNewSlide()`, которая в действительности и выводит новое изображение. Эта функция использует объект `XMLHttpRequest`, чтобы отослать запрос серверу для получения новой порции данных.

Ключевой вопрос: откуда страница ChineSites.html получает свои данные? Подходы к этому могут быть разные. Сложные решения могут вызывать какую-либо сервисную службу или сценарий PHP. Также новое содержимое может создаваться на лету или извлекаться из базы данных. В этом же примере применяется низкотехнологичное решение, которое будет работать на любом веб-сервере — выполняется поиск файла с указанным именем. Например, файл для первого слайда называется ChinaSites1_slide.html, файл для второго изображения — ChinaSites2_slide.html, и т. д. Каждый такой файл содержит только небольшой фрагмент разметки (не для всей страницы). Например, разметка файла ChinaSites5_slide.html такая:

```
<figure>
  <h2>Wishing Tree</h2>
  <figcaption>Make a wish and toss a red ribbon up into the branches of
     this tree. If it sticks, good fortune may await.</figcaption>
    // Загадайте желание и бросьте красную ленту в ветви этого дерева.
    // Если она зацепится, вас ожидает удача.
    <img src="wishing_tree.jpg">
  </figure>
```

Зная, где хранятся наши данные, мы можем с достаточной легкостью создать объект `XMLHttpRequest` для получения требуемого файла. Имя файла можно создать простой строкой кода, используя текущее значение счетчика. Полностью функция `goToNewSlide()` для запроса и получения нового слайда выглядит так:

```
var req = new XMLHttpRequest();

function goToNewSlide() {
  if (req != null) {
    // Подготавливаем запрос на файл слайда.
    req.open("GET", "ChinaSites" + slideNumber + "_slide" +
                    ".html", true);

    // Подключаем функцию для обработки данных слайда.
    req.onreadystatechange = newSlideReceived;

    // Отправляем запрос.
    req.send();
  }
}
```

Последний шаг — это скопировать полученные данные в элемент `<div>`, который отображает текущий слайд:

```
function newSlideReceived() {
  if ((req.readyState == 4) && (req.status == 200))  {
      document.getElementById("slide").innerHTML = req.responseText;
  }
}
```

Совет

Чтобы придать этому примеру больше шика, изображения можно менять, используя эффект перехода. Например, новое изображение может постепенно проявляться, в то время как старое постепенно исчезает. Для реализации этого эффекта требуется только свойство `opacity` и небольшой сценарий JavaScript, исполняющийся по таймеру. (Описания классического подхода см. по адресу **http://clagnut.com/sandbox/imagefades.php**, а описание подхода на основе стилей, использующего новую и еще не полностью поддерживаемую возможность переходов CSS3, см. здесь: **http://css3.bradshawenterprises.com/cfimg2/**.) Это одно из достоинств динамических страниц, использующих объект `XMLHttpRequest` — они могут полностью управлять способом представления содержимого.

На этом обсуждение этого примера не заканчивается. В *разд. "Управление историей просмотров" главы 12* мы используем возможность HTML5 для управления историей просмотров веб-страницы, чтобы изменять URL в соответствии с текущим слайдом. Но сейчас мы перейдем к рассмотрению двух новых способов взаимодействия с веб-сервером.

Отправляемые сервером события

Объект `XMLHttpRequest` позволяет веб-странице задать серверу вопрос и получить на него немедленный ответ. Но это обмен типа "один к одному" — после предоставления сервером ответа данный сеанс взаимодействия завершается. Веб-сервер не может повременить несколько минут и отправить странице другое сообщение с обновленной информацией.

Но некоторые типы веб-страницы могли бы извлечь пользу из более длительных сеансов взаимодействия с сервером. Возьмем, например, страницу биржевых котировок на Google Finance (**www.google.com/finance**). Если оставить это страницу открытой, то периодически мы будем видеть автоматическое обновление котировок. Другим примером будет новостная лента, например компании BBC (**www.bbc.co.uk/news**). Оставив эту страницу открытой, вы увидите, что в течение дня список передовиц периодически обновляется. Подобным образом доставляются новые сообщения в ящик поступающих сообщений любой программы электронной почты с веб-интерфейсом наподобие Hotmail (**www.hotmail.com**).

Во всех этих примерах применяется метод, называющийся *опросом* (polling). Суть его состоит в том, что веб-страница периодически (например, каждые несколько минут) запрашивает у веб-сервера новые данные. Для реализации этого механизма обновления веб-страницы используется функция JavaScript `setInterval()` или `setTimeout()` *(см. разд. "Простая анимация" главы 7)*, которая активирует код по истечении установленного периода времени.

Метод опроса является разумным решением, но иногда неэффективным. Во многих случаях это означает, что нужно вызывать веб-сервер и устанавливать новое подключение только для того, чтобы узнать, что ничего нового нет. Умножьте это на сотни или тысячи посетителей, обращающихся к веб-странице одновременно, и это может вылиться в ненужную нагрузку на ваш веб-сервер.

Одно из возможных решений этой проблемы — это использование *отправляемых сервером событий* (server-sent events), что позволяет удерживать открытым подключение к веб-серверу. Веб-сервер может отправлять странице сообщения в любое время, и при этом отсутствует необходимость постоянно отключаться и подключаться к серверу и исполнять один и тот же сценарий. (Но если в этом есть необходимость, то можно использовать и опрос, т. к. отправляемые сервером события поддерживают эту возможность.) Лучшее этой технологии в том, с какой легкостью используется система отправляемых сервером событий, а также в том, что она работает на большинстве веб-хостов и исключительно надежная. Но это сравнительно новая система с соответствующим уровнем поддержки в браузерах (табл. 11.1).

Таблица 11.1. *Поддержка браузерами системы отправляемых сервером событий*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
Минимальная версия	—	6*	5	5	11	4	—

*На момент написания этой книги доступны только ранние бета-сборки этой версии.

ПРИМЕЧАНИЕ

Если вы хотите воспользоваться заполнителем (polyfill), которым эмулирует отправляемые сервером события посредством системы опроса, несколько хороших кандидатов можно найти здесь: **http://tinyurl.com/polyfills**.

В последующих разделах мы создадим простой пример, демонстрирующий использование отправляемых сервером событий.

Формат сообщений

В отличие от объекта `XMLHttpRequest`, система отправляемых сервером событий не разрешает передавать данные в произвольном формате, а требует придерживаться простого, но установленного формата. Каждое сообщение должно начинаться текстом `data:`, за которым следует собственно текст сообщения, а в заключение — последовательность символов перехода на новую строку, которая во многих языках программирования, включая PHP, состоит из символов `\n\n`.

Вот пример текста сообщения:

```
data: Это сообщение было отправлено вам веб-сервером.\n\n
```

Сообщение разрешается разбить на несколько частей. Для этого используется последовательность символов окончания строки, которая часто состоит из одной пары `\n`. Это облегчает отправку сложных данных, наподобие следующих:

```
data: Это сообщение было отправлено вам веб-сервером.\n
data: Надеемся, вам оно понравится.\n\n
```

Обратите внимание, что каждую часть сообщения нужно начинать текстом `data:`, а все сообщение завершать признаком перехода на новую строку `\n\n`.

Этот метод можно использовать даже для отправки данных в формате JSON *(см. разд. "Сохранение объектов" главы 9)*, что позволяет преобразовать текст в объект в один прием:

```
data: {\n
data: "messageType": "statusUpdate",\n
data: "messageData": "Work in Progress"\n
data: }\n\n
```

Вместе с данными сообщения веб-сервер может отправить однозначное идентифицирующее значение (используя префикс `id:`) и время тайм-аута для подключения (используя префикс `retry:`):

```
id: 495\n
retry: 15000\n
data: Это сообщение было отправлено вам веб-сервером.\n\n
```

Веб-страница обращает внимание только на данные сообщения и игнорирует идентификатор сообщения и время тайм-аута. Этими подробностями занимается браузер. Например, прочитав предыдущее сообщение, браузер знает, если он потеряет подключение к веб-серверу, то должен попытаться подключиться повторно после 15 000 мс (т. е. 15 секунд). При повторном подключении браузер также должен отправить номер идентификатора 495, чтобы сервер мог его распознать.

ПРИМЕЧАНИЕ

Веб-страница может потерять подключение к веб-серверу по множеству разных причин, включая кратковременный сетевой сбой или истечение времени ожидания данных прокси-сервером. Если возможно, браузер будет пытаться восстановить подключение автоматически, выждав по умолчанию 3 секунды.

Отправка сообщений с помощью серверного сценария

Зная формат сообщений, мы можем с легкостью создать серверный код для их отправки. Опять же, для создания простого примера, который будет поддерживаться практически всеми веб-хостами, удобно использовать язык серверных сценариев PHP. На рис. 11.4 показана страница, которая получает регулярные сообщения от сервера. В данном случае сообщения предназначены просто для демонстрационных целей и содержат лишь текущее время на веб-сервере.

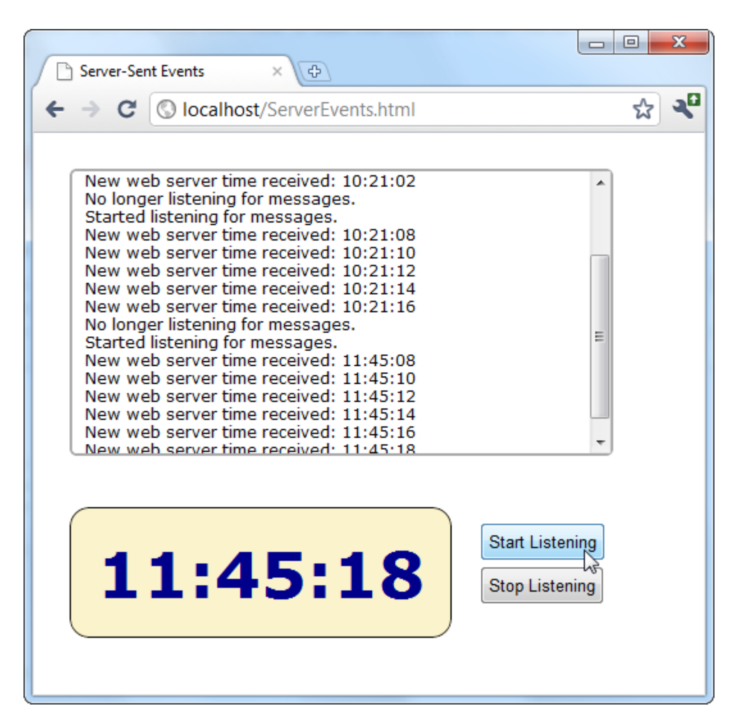


Рис. 11.4. Когда эта страница находится в режиме прослушивания, она получает постоянный поток сообщений от веб-сервера — приблизительно одно сообщение каждые две секунды. Каждое сообщение добавляется в верх прокручиваемого списка, а табло внизу отображает время получения последнего сообщения

ПРИМЕЧАНИЕ

В данном примере время на веб-сервере единственно, что постоянно обновляется. Это хорошо служит целям демонстрации принципов работы системы отправляемых сервером событий. Но в настоящем приложении сообщения содержали бы что-то более ценное, например последние поступления для новостной ленты.

Серверная часть этого примера просто информирует о текущем времени по регулярным интервалам. Весь код PHP для этого выглядит так:

```
<?php
  header("Content-Type: text/event-stream");
  header('Cache-Control: no-cache');

  // Запускаем бесконечный цикл.
  do {
    // Получаем текущее время.
    $currentTime = date("h:i:s" time());

    // Отправляем полученное время в сообщении.
    echo "data: " . $currentTime . PHP_EOL;
    echo PHP_EOL;
    flush();
```

```
    // Ожидаем 2 секунды перед тем, как создавать новое сообщение.
    sleep(2);
  } while(true);
?>
```

В начале этого сценария устанавливаются два важных заголовка. Сначала MIME-типу присваивается значение `text/event-stream`, что требуется стандартом для отправляемых сервером событий:

```
header("Content-Type: text/event-stream");
```

Потом веб-серверу (а также прокси-серверам) дается указание отключить кэширование. Если этого не сделать, то сообщения могут прибывать в пакетах по нерегулярным интервалам.

```
header('Cache-Control: no-cache');
```

Остальная часть кода состоит из бесконечного цикла (или, по крайней мере, исполняющегося до тех пор, пока не исчезнет клиент). При каждом исполнении цикла встроенная функция `time()` получает текущее время (в формате `часы:минуты:секунды`) и присваивает это значение переменной `$currentTime`:

```
$currentTime = date("h:i:s", time());
```

Далее эта информация используется для создания сообщения в правильном формате, которое потом отправляется посредством команды `echo`. В данном примере сообщение состоит из одной строки, начинающейся текстом `data:`, за которым следуют данные времени. Сообщение завершается константой `PHP_OEL`, которая представляет комбинацию символов `\n`, обозначающих конец строки:

```
echo "data: " . $currentTime . PHP_EOL;
echo PHP_EOL;
```

ПРИМЕЧАНИЕ

Если этот код выглядит несколько странно, так это, наверное, потому, что в PHP для конкатенации строк используется оператор в виде точки (`.`). Этот оператор работает таким же образом, как и оператор в виде знака "плюс" (+) с текстом в JavaScript, только его нельзя случайно перепутать с операцией математического сложения.

Функция `flush()` обеспечивает немедленную отправку данных, а не помещение их в буфер для отправки после завершения выполнения кода PHP. Наконец, функция `sleep()` приостанавливает исполнение кода на две секунды, после чего начинается исполнение новой итерации цикла.

СОВЕТ

Если установить слишком большое время между сообщениями, *прокси-сервер* (сервер, который расположен между веб-сервером и компьютером клиента и управляет трафиком) может разорвать подключение. Чтобы предотвратить это, можно отправлять незначащее сообщение в виде символа комментария (простое двоеточие — :) без текста приблизительно каждые 15 секунд.

Обработка сообщений в веб-странице

Создание веб-страницы для обработки отправляемых сообщений даже проще, чем кода для отправки этих сообщений. Блок <body> страницы разделяется на три блока <div> — один для окна списка сообщений, другой для табло для отображения текущего времени и один для кнопок запуска и остановки процесса:

```
<div id = "messageLog"></div>
<div id = "timeDisplay"></div>

<div id="controls">
  <button onclick=" startListening()">Start Listening</button><br>
  <button onclick="stopListening()">Stop Listening</button>
</div>
```

При загрузке страница находит элементы `messageLog` и `timeDisplay` и сохраняет их в глобальных переменных, что обеспечивает всем нашим функциям легкий доступ к ним:

```
var messageLog;
var timeDisplay;

window.onload = function() {
  messageLog = document.getElementById("messageLog");
  timeDisplay = document.getElementById("timeDisplay");
};
```

Процесс отправки сообщений веб-сервером и получения их страницей начинается нажатием кнопки **Start Listening**. В это время код создает новый объект `EventSource`, предоставляя URL серверного ресурса, который будет отправлять сообщения. (В данном примере это PHP-сценарий TimeEvents.php.) Потом к событию `onMessage` подключается функция `startListening()`, которая срабатывает при каждом получении сообщения страницей:

```
var source;

function startListening()  {
  source = new EventSource("TimeEvents.php");
  source.onmessage = receiveMessage;
  messageLog.innerHTML += "<br>" + "Started listening for messages.";
}
```

Совет

Выяснить, поддерживает ли веб-сервер систему отправляемых сервером сообщений, можно, выполнив проверку на существование свойства `window.EventSource`. Если это свойство не существует, то нужно будет прибегнуть к резервному решению. Например, можно использовать объект `XMLHttpRequest`, чтобы периодически обращаться к веб-серверу для получения данных.

Сообщение можно получить из свойства `data` объекта `event` при активировании события `receiveMessage`. В данном примере свойство `data` добавляет новое сообщение в список сообщений и обновляет табло часов:

```
function receiveMessage(e) {
  messageLog.innerHTML += "<br>" + "New web server time received: " +
                                    e.data;
  timeDisplay.innerHTML = e.data;
}
```

Обратите внимание, что из сообщения удалена вся служебная информация (текст `data:` и признак перехода на новую строку `/n/n`), и отображается только требуемое содержимое.

Наконец, прослушивание событий сервера страницей можно прекратить в любое время, вызвав метод `close()` объекта `EventSource`. Делается это так:

```
function stopListening() {
  source.close();
  messageLog.innerHTML += "<br>" + "No longer listening for messages.";
}
```

Опрос посредством серверных событий

В предыдущем примере мы рассмотрели наиболее простое использование системы серверных событий, которое, по большому счету, сводится к следующему: страница отправляет запрос, подключение остается открытым, и сервер периодически отправляет странице информацию. Браузеру может потребоваться выполнить повторное подключение (что он делает автоматически), но только в случае проблемы с подключением или же если оно было временно прервано по другим причинам (например, с целью экономии заряда батареи на мобильном устройстве).

Но что произойдет, если прекратится исполнение сценария на веб-сервере, и последний закроет подключение? Довольно интересно, хотя отключение выполняется не вследствие неполадок, а в результате целенаправленных действий веб-сервера, веб-страница автоматически возобновляет подключение (после ожидания, по умолчанию равному 3 секундам) и снова запрашивает данный сценарий, начиная его исполнение с самого начала.

Это поведение можно обернуть в свою пользу. Допустим, что мы создали сравнительно небольшой серверный сценарий, который отправляет только одно сообщение. Теперь действия веб-страницы похожи на действия при опросе *(см. разд. "Отправляемые сервером события" ранее в этой главе)* в том, что она периодически восстанавливает подключение. Разница состоит лишь в том, что период между подключениями устанавливается веб-сервером, а не JavaScript-кодом страницы, как делается при использовании традиционного опроса.

В следующем примере используется комбинированный подход. Серверный сценарий удерживает подключение открытым и периодически отправляет сообщения в

течение 1 минуты. Потом сценарий дает указание браузеру подключиться через 2 минуты и закрывает подключение:

```
<?php
  header("Content-Type: text/event-stream");
  header('Cache-Control: no-cache');

  // Указываем браузеру ожидать 2 минуты после закрытия подключения,
  // прежде чем подключаться опять.
  echo "retry: 120000" . PHP_EOL;

  // Сохраняем время начала.
  $startTime = time();

  do {
    // Отправляем сообщение.
    $currentTime = date("h:i:s", time());
    echo "data: " . $currentTime . PHP_EOL;
    echo PHP_EOL;
    flush();

    // Если прошла минута со времени начала сценария, завершаем
    // его исполнение.
    if ((time() — $startTime) > 60) {
         die();
    }

    // Ожидаем 5 секунд и отправляем новое сообщение.
    sleep(5);
  } while(true);
?>
```

Теперь при просмотре страницы мы будем получать регулярные сообщения в течение 1 минуты, после чего следует перерыв в 2 минуты, после чего процесс повторяется (рис. 11.5). В настоящем приложении при закрытии подключения веб-сервер может отправить браузеру специальное сообщение, информирующее, что больше нет причин ожидать обновленные данные (например, потому что фондовые биржи закрылись). Тогда веб-страница могла бы прекратить процесс, вызвав метод `close()` объекта `EventSource`.

Примечание

В случае сложных серверных сценариев автоматическое повторное подключение браузера может работать не совсем так, как ожидалось. Например, подключение может быть прервано посредине исполнения веб-сервером какой-либо задачи. В таком случае серверный сценарий может отправить всем клиентам идентификатор (см. последние два абзаца *разд. "Формат сообщений" ранее в этой главе*), который браузер вернет серверу при повторном подключении. Но ответственность за создание подходящего идентификатора, отслеживание соответствия каждого идентификатора с его

задачей (например, сохраняя определенную информацию в базе данных), а потом попытку возобновить процесс с того момента, когда он был прерван, лежит на серверном коде. А все эти задачи могут быть очень проблематичными, если ваши навыки кодирования не на высшем уровне.

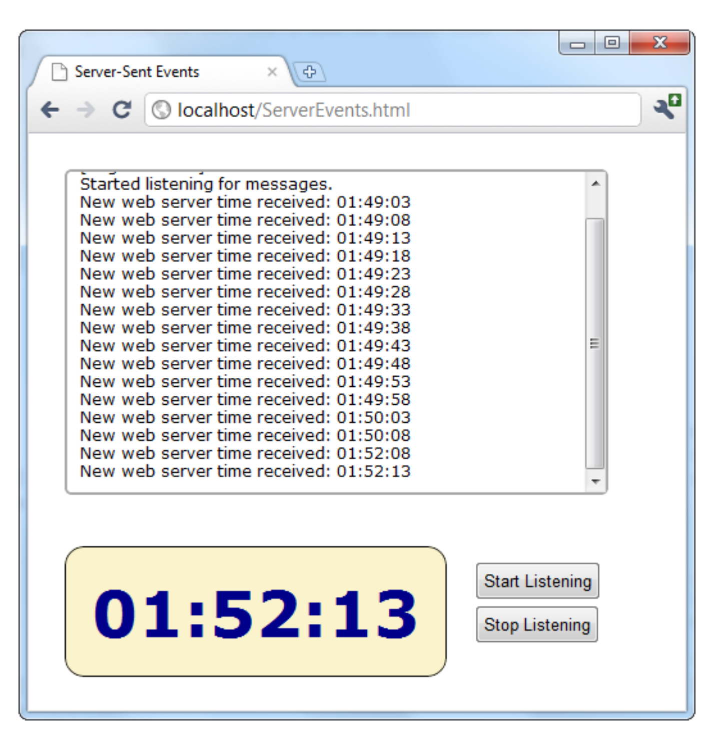


Рис. 11.5. В этой странице используется комбинация отправки потоковых сообщений (в течение одной минуты) и опроса (выполняемого в течение двух минут после завершения потоковой части). Такой подход удобен, когда требуется свести к минимуму трафик веб-сервера, в зависимости от частоты обновления данных и от того, насколько важно иметь самые свежие данные

Веб-сокеты

Серверные события являются идеальным инструментом, когда требуется получить последовательность сообщений с веб-сервера. Но при этом связь получается полностью односторонней. Браузер не может отвечать на сообщения или вступать в более сложный диалог с сервером.

Если вы создаете веб-приложение, в котором требуется серьезное двустороннее взаимодействие браузера с веб-сервером, лучшим подходом к его реализации (не прибегая к помощи Flash) будет, возможно, использование объекта `XMLHttpRequest`. В зависимости от типа создаваемого приложения этот подход может работать так, как требуется. Но здесь существует и достаточное количество возможных проблем. Прежде всего, объект `XMLHttpRequest` не очень хорошо подходит для быстрого обмена множественными сообщениями (например, в чате). Потом, в нем нет возможности связать один вызов с другим, поэтому при каждом новом запросе от веб-

страницы сервер должен вычислять с самого начала, кому эта страница принадлежит. Поэтому уровень сложности кода для обработки ряда связанных запросов от веб-страницы может очень быстро вырасти до практически нереализуемой.

Для всех этих проблем есть решение, хотя оно еще не вполне готово. Этим решением является технология *веб-сокетов* (web sockets), которая позволяет браузеру удерживать открытое подключение к серверу и обмениваться сообщениями в течение любого требуемого времени. Технология веб-сокетов вызвала большое возбуждение в среде веб-разработчиков, но она еще находится в процессе развития и не имеет хорошего уровня браузерной поддержки. Изначально поддержка веб-сокетов была добавлена в браузеры Firefox 4 и Opera 11, но была удалена через несколько месяцев в связи с проблемой потенциальных нарушений безопасности. Но эту возможность планируется снова ввести в Firefox 6, используя доработанную версию первоначального протокола; также она будет, скорее всего, снова добавлена в будущие версии Opera. Корпорация Microsoft еще не приняла никаких твердых решений, но публично экспериментирует с этой возможностью (вы можете испытать ее сами в тестовой лаборатории по адресу **http://tinyurl.com/3szzz72**).

В связи с этими постоянными изменениями в стандарте веб-сокетов, о нем трудно сказать что-либо определенное. Более того, для него даже не приводится таблица браузерной поддержки, как для других возможностей, т. к. разные версии браузеров поддерживают различные версии этого стандарта, которые обычно несовместимы друг с другом. Это означает, что сервер веб-сокетов, предназначенный для работы с одной версией веб-сокетов, не будет работать с браузерами, поддерживающими другую.

Примечание

На данный момент лучше всего тестировать страницы, использующие веб-сокеты, в браузере Chrome, который предоставляет наиболее последовательную поддержку для них. Можно также попробовать Firefox, если сможете найти бета-версию Firefox 6. (Технически можно включить поддержку сокетов в более ранних версиях Firefox, используя скрытый параметр. Но эта функциональность довольно неуклюжа и использует устаревшую версию стандарта веб-сокетов, так что вряд ли с ней стоит морочиться.)

Получение доступа к веб-сокетам

Если вы добрались до этих строк, то знаете, что вас не испугают возможности настолько новые, что их стандарт еще не был полностью определен, а они сами полностью не реализованы в браузерах. Но вопрос заключается в том, сколько времени вам следует затратить на изучение веб-сокетов в настоящее время, иными словами, насколько хороши перспективы, что они выльются в ценную возможность веб-программирования в будущем, и следует ли ожидать, что через год или два их можно будет использовать в своих веб-приложениях?

Прежде чем ответить на этот вопрос, вам нужно понять две важные вещи. Первая — веб-сокеты являются специализированным инструментом. Они актуальны для таких приложений, как чат, массивные многопользовательские игры или инструмент для пирингового взаимодействия. Веб-сокеты позволяют создавать новые

типы приложений, но применять их в большинстве современных веб-приложений, движимых JavaScript, скорее всего, не имеет смысла.

Вторая — решения на основе веб-сокетов могут быть чрезвычайно сложны. Разработать JavaScript-код для одной страницы будет достаточно простой задачей. Но для создания серверного приложения вам потребуются бешеные знания и навыки программирования, включая понимание концептов многопотоковости и сетевого взаимодействия.

Для использования веб-сокетов на веб-сервере вашего сайта должна исполняться специальная программа, которая будет, как ожидается, называться сервером веб-сокетов. На эту программу возлагается ответственность за координирование взаимодействия всех участников, и после запуска она работает безостановочно.

Примечание

Многие хостинговые компании не допускают долго работающих программ, если только вы не оплатите выделенный веб-сервер, т. е. сервер, обслуживающий лишь ваш сайт. Если у вас обычный общий хостинг, вы, скорее всего, не сможете размещать на нем страницы, в которых используются веб-сокеты. Даже если вы умудритесь запустить сервер веб-сокетов и удерживать его в рабочем состоянии, владелец вашего хостинга, скорее всего, выявит и выключит его.

Чтобы дать вам представление о масштабе сервера веб-сокетов, рассмотрите некоторые из задач, которые сервер сокетов должен выполнять:

- составить "словарь" сообщений, иными словами, решить, какие типы сообщений являются допустимыми, а какие — нет;
- содержать список всех текущих подключенных клиентов;
- выявлять ошибки при отправке сообщений клиентам и прекратить попытки связаться с ними, если кажется, что их больше не существует;
- обрабатывать все данные в оперативной памяти, т. е. данные, доступ к которым может потребоваться всем клиентам, и делать это надежно и безопасно. Здесь имеется обилие возможных неявных проблем, например, когда один клиент пытается присоединиться к обмену, в то время как другой отключается, а информация об обоих хранится в одном и том же объекте в памяти.

Разработчики, скорее всего, никогда не будут сами создавать серверную программу, использующую веб-сокеты, т. к. это просто-напросто не стоит требуемых для этого значительных усилий. Самым легким подходом в этой области будет установить чей-то другой сервер веб-сокетов и разрабатывать свои веб-страницы под него. Так как использование части JavaScript стандарта веб-сокетов не несет трудностей, это не должно доставлять каких-либо проблем. Другим подходом будет взять чей-либо код сервера веб-сокетов и подогнать его под свои требования. В настоящее время существует великое множество проектов (многие из которых бесплатные и с открытым кодом), в которых разрабатываются серверы веб-сокетов для решения различных задач, на разных языках серверного программирования. Подробности *см. во врезке "Практические занятия для опытных пользователей. Серверы веб-сокетов" в конце этой главы*.

Простой клиент веб-сокетов

С точки зрения веб-страницы функциональность веб-сокетов легко понять и использовать. Первый шаг — это создать объект `WebSocket` и передать ему URL. Код для этого подобно следующему:

```
var socket = new WebSocket("ws://localhost/socketServer.php");
```

Строка URL начинается с текста `ws://`, который идентифицирует подключение типа веб-сокет. Этот URL указывает файл веб-приложения на сервере (в данном случае это сценарий socketServer.php). Стандарт веб-сокетов также поддерживает URL, которые начинаются с текста `wss://`, что указывает на требование использовать безопасное, зашифрованное подключение (точно так же, как и при запросе веб-страницы указывается URL, начинающийся с `https://` вместо `http://`).

> ***Примечание***
>
> Веб-сокеты могут подключаться не только к своему веб-серверу. Веб-страница может открыть подключение к серверу веб-сокетов, исполняющемуся на другом веб-сервере, не требуя для этого никаких дополнительных усилий.

Само обстоятельство создания объекта `WebSocket` понуждает страницу пытаться подключиться к серверу. Дальше надо использовать одно из четырех событий объекта `WebSocket`: `onOpen` (при установлении подключения), `onError` (когда возникает ошибка), `onClose` (при закрытии подключения) и `onMessage` (когда страница получает сообщение от сервера):

```
socket.onopen = connectionOpen;
socket.onmessage = messageReceived;
socket.onerror = errorOccurred;
socket.onopen = connectionClosed;
```

Например, в случае успешного подключения неплохо бы отправить соответствующее подтверждающее сообщение. Такое сообщение доставляется с помощью метода `send()` объекта `WebSocket`, которому в качестве параметра передается обычный текст. Далее приведена функция, которая обрабатывает событие `onOpen` и отправляет сообщение:

```
function connectionOpen() {
  socket.send("UserName:jerryCradivo23@gmail.com");
}
```

Предположительно, веб-сервер получит это сообщение и даст на него ответ.

События `onError` и `onClose` можно использовать для отправки извещений посетителю веб-страницы. Но безоговорочно самым важным является событие `onMessage`, которое срабатывает при получении новых данных от сервера. Опять же, код JavaScript для обработки этого события не представляет никаких сложностей — мы просто извлекаем текст сообщения из свойства `data`:

```
function messageReceived(e)  {
  messageLog.innerHTML += "<br>" + "Message received: " + e.data;
}
```

Если веб-страница решит, что вся ее работа выполнена, она может закрыть подключение, используя метод `disconnect()`:

```
socket.disconnect();
```

Из этого обзора веб-сокетов можно видеть, что использование сервера веб-сокетов стороннего разработчика не представляет никаких трудностей — нам нужно лишь знать, какие сообщения отправлять, а какие — ожидать.

ПРИМЕЧАНИЕ

Чтобы заставить подключение веб-сокетов работать, выполняется большой объем работы за кулисами. Прежде всего, веб-страница устанавливает связь по обычному стандарту HTTP. Потом это подключение нужно повысить до подключения веб-сокетов, позволяющего свободную двустороннюю связь. На этом этапе возможны проблемы, если между компьютером клиента и веб-сервером находится прокси-сервер (как, например, в типичной корпоративной сети). Прокси-сервер может отказаться сотрудничать и разорвет подключение. Эту проблему можно решить, обнаруживая неудачное подключение (посредством события `onError` объекта `WebSocket`) и применяя один из заполнителей (polyfills) для сокетов, описанных на веб-сайте GitHub (**http://tinyurl.com/polyfills**). Эти заполнители применяют метод опроса, чтобы эмулировать подключение веб-сокетов.

Примеры веб-сокетов в сети

Если вы заинтересованы опробовать веб-сокеты, в сети есть много сайтов, на которых можно запустить свою разработку.

Для начала попробуйте сайт **http://websocket.org/echo.html**, который предоставляет простейший сервер веб-сокетов: веб-страница отправляет ему сообщение, а он возвращает это же сообщение веб-странице (рис. 11.6).

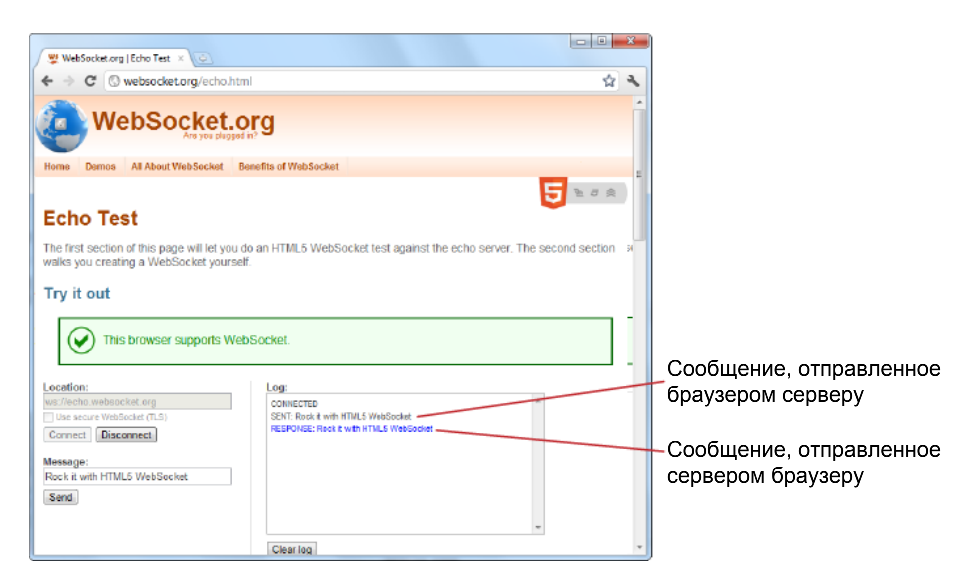


Рис. 11.6. Сервер веб-сокетов, который просто отправляет назад полученные сообщения, не выиграет никаких призов за инновационные разработки в программном обеспечении. Но он дает возможность убедиться, насколько легко взаимодействовать с готовым сервером веб-сокетов

Хотя этот сервер веб-сокетов и не представляет ничего особенного, на нем вы можете испробовать все возможности объекта `WebSocket`. Более того, к этому серверу можно подключиться со страницы, расположенной как на промышленном веб-сервере, так и на тестовом веб-сервере на вашем компьютере, или даже со страницы, просто запускаемой с жесткого диска.

Существуют и серверы веб-сокетов, предоставляющие другие возможности, включая следующие.

- **Простой чат.** Чат, в котором все разговаривают со всеми. Отправляемые сообщения получают все участники чата. Находится по этому адресу: **http://html5demos.com/web-socket**.
- **Многопользовательский альбом.** Эта страница объединяет веб-сокеты с холстом HTML5. То, что вы рисуете на своем холсте, отображается на холсте других участников, и наоборот. Простой концепт, но очень впечатляющий на практике. Находится по этому адресу: http://mrdoob.com/projects/multiuserpad.

ПРАКТИЧЕСКИЕ ЗАНЯТИЯ ДЛЯ ОПЫТНЫХ ПОЛЬЗОВАТЕЛЕЙ
Серверы веб-сокетов

Чтобы иметь возможность испытать свой проект по веб-сокетам, вам нужен сервер веб-сокетов, с которым ваша страница могла бы общаться. Хотя рассмотрение кода сервера веб-сокетов (объемом, по крайней мере, в несколько десятков строк) выходит за рамки этой книги, тестовый сервер можно найти во многих местах.

Среди прочих, можно порекомендовать следующие.

- **PHP.** Этот простой и слегка сыроватый проект будет хорошей отправной точкой для создания сервера веб-сокетов на PHP. Загрузить его можно здесь: **http://code.google.com/p/phpwebsocket**.
- **Ruby.** Существует несколько образцов сервера веб-сокетов на Ruby, но этот, применяющий модель "Event — Machine", пользуется особенной популярностью. Загрузить его можно здесь: **http://github.com/igrigorik/em-websocket**.
- **Python.** Сервер веб-сокетов в виде модуля расширения для Apache на языке Python. Загрузить его можно здесь: **http://code.google.com/p/pywebsocket**.
- **.NET.** Назвать простым этот всеохватывающий проект нельзя. Но он содержит завершенный сервер веб-сокетов на языке C# на основе платформы .NET корпорации Microsoft. Загрузить его можно по адресу **http://superwebsocket.codeplex.com**.
- **Java.** По своему масштабу этот проект похож на проект .NET, но чисто на языке Java. Загрузить его можно здесь: **http://jwebsocket.org**.
- **node.JS.** В зависимости от того, кого вы спросите, система node.JS для разработки веб-приложений на JavaScript — это либо одна из наиболее перспективных платформ, либо просто разросшийся тестовый инструмент. В любом случае, сервер веб-сокетов для этой платформы можно загрузить здесь: **http://github.com/miksago/node-websocket-server**.
- **Kaazing.** В отличие от других пунктов этого списка, Kaazing не предоставляет кода для сервера веб-сокетов. Это развитый сервер веб-сокетов, который можно лицензировать для своего веб-сайта. Для разработчиков, которые предпочитают делать все своими руками, он не будет представлять интереса. Но удобно использовать его на менее амбициозных веб-сайтах, особенно принимая во внимание то обстоятельство, что он содержит встроенную поддержку резервных решений в своих клиентских библиотеках (которые сначала пытаются применить стандарт веб-сокетов HTML5, затем Flash, а потом опрос посредством сценариев на JavaScript). Дополнительную информацию см. здесь: **http://kaazing.com/products/html5-edition.html**.

12

ГЛАВА 12

Несколько полезных возможностей на JavaScript

На данный момент мы рассмотрели все ключевые темы HTML5. Мы применили возможности этого стандарта для написания более значащей и лучше структурированной разметки, познакомились с его расширенными графическими функциональностями, такими как видео и динамическое рисование, а также применили их для создания самодостаточных страниц, движимых JavaScript, которые могут продолжать работать даже при отсутствии подключения к Интернету.

В этой главе мы рассмотрим еще три возможности, которые представляют значительный интерес. Подобно большей части уже рассмотренного материала, эти возможности расширяют диапазон функциональностей веб-страницы, и все это посредством небольшого объема кода JavaScript. Далее перечислены темы, которые нам предстоит изучить.

- **Геолокация.** Хотя эта возможность часто рассматривается как часть HTML5, в действительности геолокация является отдельным стандартом, который никогда не был в ведении группы WHATWG *(см. разд. "HTML5: возвращение к жизни" главы 1)*. Но подобно многим другим возможностям HTML5 геолокация позволяет создавать более мощные веб-страницы с помощью кода JavaScript. В частности, используя геолокацию, можно получить довольно важную информацию: географические координаты посетителя страницы.
- **Фоновые вычисления.** По мере того как веб-разработчики создают более интеллектуальные страницы, для которых применяются все большие объемы кода JavaScript, очень важно исполнять определенные задачи в фоновом режиме тихо, не привлекая внимания, и в течение длительного периода времени. Для решения этой задачи *можно* было бы использовать таймеры и другие приемы, но возможность фоновых вычислений предполагает более удобное решение.
- **История сеансов.** В старые добрые времена веб-страница выполняла одну задачу — отображала содержимое. В результате посетителям приходилось тратить много времени, щелкая по ссылкам, чтобы перейти с одного документа на другой. Но сегодня снабженная JavaScript-кодом страница может загружать содержимое с другой страницы, не вызывая при этом полного обновления. Таким об-

разом, JavaScript позволяет создать более плавное зрительное восприятие страницы посетителем. Но добавление этой возможности также доставляет новые проблемы, такие как, например, необходимость синхронизации URL с текущим содержимым. Веб-разработчики применяют множество продвинутых методов, чтобы содержать свои приложения в порядке. Теперь HTML5 предоставляет средство для работы с историей сеансов, которое готово помочь им с этой задачей (по крайней мере, иногда).

ПРИМЕЧАНИЕ

По мере углубления в изучение этих трех возможностей мы получим еще лучшее представление о размахе стандарта HTML5. То, что вначале выглядело как просто несколько хороших идей, втиснутых в слишком амбициозный стандарт, переросло во множество новых возможностей, решающих целый диапазон разных задач, и все это удерживается вместе с помощью всего лишь нескольких ключевых концептов (таких как семантика, JavaScript и CSS3).

Геолокация

Геолокация позволяет определять географическое местоположение посетителей веб-сайта. И это не означает просто страну или даже город, в котором находится посетитель. Геолокация может сузить информацию о местоположении человека к городскому кварталу или даже определить его точные координаты, если он зашел на страницу со смартфона.

Все основные браузеры предоставляют хороший уровень поддержки геолокации (табл. 12.1).

Таблица 12.1. *Браузерная поддержка возможности геолокации*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
Минимальная версия	9	3.5	5	5	10.6	3.2	2.1

Тем не менее в этой поддержке есть слабые места в виде старых версий Internet Explorer, таких как IE 7 и IE 8. Одним из способов предоставить поддержку геолокации в этих версиях IE будет создать резервное решение, использующее Google Gears (**http://code.google.com/apis/gears**). Проект Google Gears был разработан до HTML5 и нацелен на предоставление многих из одинаковых с этим стандартом возможностей, включая подобную систему геолокации. Хотя проект Google Gears в настоящее время не рекомендуется к использованию (в Google прекратили работать над ним и не предоставляют поддержки для новых версий браузеров), он сгодится для старых версий обозревателей. Недостаток этого подхода состоит в том, что, как и все подключаемые модули, модуль Google Gears должен быть установлен на компьютере посетителя. Если на компьютере не установлен модуль Google Gears, можно использовать модуль Chrome Frame *(см. врезку "Малоизвестная или недооценен-*

ная возможность. Модифицирование IE с помощью Google Chrome Frame" в конце главы 1), или же просто запросить посетителей ввести свое местонахождение.

> **ПРИМЕЧАНИЕ**
>
> Большинство новых возможностей JavaScript, которые мы рассмотрели в этой книге, были частью первоначальной спецификации HTML5, а потом были отделены, когда спецификацию передали в ведение организации W3C. Но геолокация никогда не была частью HTML5, а просто полностью сформировалась приблизительно в то же самое время. Но почти все рассматривают эту возможность вместе с возможностями HTML5, как часть одной большой волны будущих технологий.

Принцип работы геолокации

Возможность геолокации вызывает довольно много вопросов у людей, которых обычно нельзя назвать параноиками. Например, каким образом может какое-то программное обеспечение знать, что я не болею, как я сказал своему начальнику, а смотрю футбол в кафе? В нем что, есть какой-то скрытый код, который отслеживает все мои действия? Интересно, что это за фургон там на улице и те люди, которые прикидываются, будто бы они меняют колесо?

К счастью, возможность геолокации не из разряда страшилок о слежке. Это потому, что даже если браузер может вычислить ваше местонахождение, он не сообщит его веб-сайту без вашего явного на то разрешения (рис. 12.1).

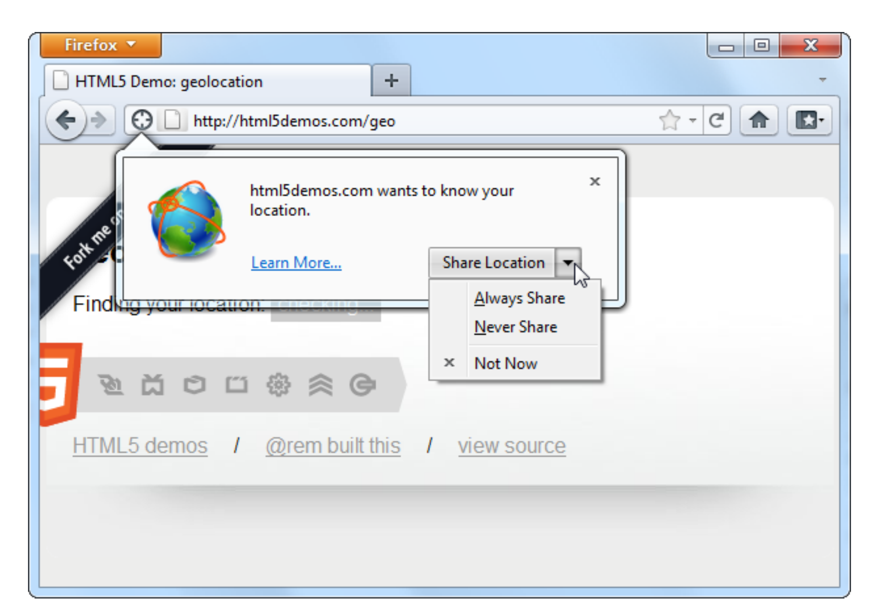


Рис. 12.1. Веб-страница хочет получить данные о местоположении посетителя. Посетитель может разрешить всегда предоставлять эти данные (**Always Share**), предоставить их только в этот раз (**Share Location**) или же никогда не предоставлять их (**Never Share**). Такое поведение браузера Firefox не является собственной инициативой его разработчиков, а официальным правилом стандарта геолокации, требующим, чтобы каждый веб-сайт, пытающийся получить данные о местоположении посетителя, получил для этого его разрешение

Чтобы вычислить местоположение посетителя, браузер заручается помощью *поставщика услуг определения местоположения* (location provider), например, для Firefox это сервис Google Location Services. Определение местоположения является задачей не из легких, и поставщик применяет несколько разных подходов для ее решения.

В случае настольного компьютера с фиксированным (т. е. небеспроводным) подключением, метод простой, но дает не очень точные результаты. Когда кто-то подключается к Интернету, данные с его компьютера или локальной сети направляются (через кабель, выделенную линию или коммутируемое телефонное соединение) на мощное сетевое аппаратное устройство, которое, в свою очередь, направляет их в Интернет. Это аппаратное устройство имеет однозначный IP-адрес, т. е. числовой код, который идентифицирует это устройство для других участников сети. Это устройство также имеет физический почтовый адрес.

Примечание

Если у вас есть немного сетевого опыта, вы должны знать, что, подобно всем другим компьютерам в локальной сети, ваш компьютер имеет свой IP-адрес. Но этот IP-адрес является частным и предназначен для того, чтобы можно отличить друг от друга разные устройства локальной сети, которые используют одно и то же подключение к Интернету (например, настольный компьютер в рабочем кабинете и лэптоп в спальне). Для целей геолокации этот адрес не используется.

Поставщик местоположения использует эти два типа информации для определения географического местоположения посетителя веб-страницы. Сначала вычисляется IP-адрес устройства, через которое осуществляется подключение, а затем определяется его физический адрес. Естественно, такой непрямой подход позволяет определить точное местоположение не пользователя, а только его интернет-провайдера. Тем не менее часто даже такие неточные результаты являются полезными. Например, если вы используете средство геолокации, чтобы найти магазин, продающий пиццу на вынос, вы можете быстро перейти в район, который вас действительно интересует — вблизи вашего дома — даже если вы находитесь не совсем рядом.

Примечание

Метод определения местоположения посредством IP-адреса является самым неточным способом геолокации. Если имеется лучший источник данных, поставщик местоположения будет использовать этот источник.

Если посетитель зашел на страницу с лэптопа или смартфона по беспроводному подключению, поставщик местоположения использует ближайшие точки беспроводного доступа. В идеальном случае поставщик местоположения выбирает данные из огромной базы данных, чтобы определить точное местонахождение этих точек доступа, а потом использует полученную информацию, чтобы определить местоположение посетителя методом триангуляции[1].

[1] Метод создания сети по опорным точкам. Построение базируется на треугольниках. — *Ред.*

А в случае подключения с мобильного телефона поставщик местоположения применяет подобный метод триангуляции, но использует сигналы от разных антенн мобильной связи. Эта быстрая, сравнительно эффективная процедура обычно позволяет определить местоположение посетителя с точностью до 1 км. (Индустриальные зоны, например центр города, имеют большее количество антенн мобильной связи, что позволяет определить местоположение с большей точностью.)

Наконец, многие мобильные устройства оснащены специальными аппаратными средствами GPS (Global Positioning Service, глобальная система навигации и определения положения), что позволяет определять местоположение таких устройств с точностью всего лишь до нескольких метров. Но этот метод геолокации имеет свой недостаток — он медленнее и потребляет больше энергии, что важно для устройств, работающих на аккумуляторах. Кроме этого, он не особенно хорошо работает в городах с большими и высокими зданиями по причине отражения сигнала от строений. Как мы увидим в *разд. "Установка параметров геолокации" далее в этой главе*, решение, использовать или нет высокоточное определение местоположения посредством GPS (если эта возможность доступна), принимается разработчиком приложения.

Конечно же, возможны и другие способы определения местоположения. Ничто не препятствует поставщику местоположения использовать для этого другие источники информации, такие как данные от RFID-устройств (Radio Frequency Identification, радиочастотная идентификация), данные от устройств Bluetooth, файлы cookies с информацией от картографического сайта наподобие Google Maps и т. п.

Совет

Местоположение, определенное поставщиком, можно также откорректировать с помощью другого средства. Например, для браузера Firefox можно использовать модуль расширения Geolocater (**http://addons.mozilla.org/en-us/firefox/addon/geolocater**), позволяющий пользователю установить местоположение, которое браузер должен указывать при просмотре веб-сайта, применяющего геолокацию. Пользователи могут применить этот метод, чтобы подделать свое местоположение, например, что компьютер, на котором посетитель просматривает веб-сайт из Простоквашино, на самом деле находится в Нью-Васюках.

Из всего этого можно сделать следующий вывод: независимо от способа подключения к Интернету, даже если посетитель веб-сайта сидит за настольным компьютером, возможность геолокации позволяет определить его местонахождение с большей или меньшей точностью. А если он выходит в Интернет с мобильного телефона или с устройства, оснащенного аппаратным обеспечением GPS, его местоположение может быть определено с точностью скорее большей, чем меньшей.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

В каких сферах применять геолокацию?

Ответив на большой вопрос, как работает геолокация, нам нужно разобраться еще с одним: какая нам от нее польза?

Здесь ключевым аспектом, который нужно понимать, является то, что функциональность геолокации позволяет определить *приблизительные* географические координа-

ты посетителя страницы. И это все. Веб-разработчик должен объединить эту простую, но важную информацию с более подробными данными о местоположении посетителя. Эти данные можно получить от веб-сервера (обычно из огромной серверной базы данных) или какой-либо географической веб-службы (скажем, Google Maps).

Например, для большого предприятия, имеющего филиалы в нескольких точках, может потребоваться сравнить местоположение посетителя веб-страницы с местоположением своих разных филиалов, чтобы определить ближайший к посетителю. А разработчик какого-либо средства социальных сетей может создать диаграмму расположения группы пользователей, показывающую им расстояние между ними. Или же данные о местонахождении можно использовать для предоставления посетителям веб-сайта какой-либо услуги, например нахождения ближайшего работающего круглосуточно магазина или кафе. В любом случае, информация о местоположении посетителя веб-страницы важна только в комбинации с другими географическими данными.

Хотя картографические и географические услуги, предоставляемые другими компаниями, лежат вне рамок этой книги, мы рассмотрим одну из них — Google Maps — в *разд. "Отображение карты" далее в этой главе*.

Определение координат посетителя

Возможность геолокации предельно проста. Она состоит из трех методов объекта `navigator.geolocation`: `getCurrentPosition()`, `watchPosition()` и `clearWatch()`.

> ***ПРИМЕЧАНИЕ***
>
> Объект `navigator` — это сравнительно незначительная часть JavaScript. Его несколько свойств предоставляют информацию о текущем браузере и его возможностях. Наиболее полезным из них является свойство `navigator.userAgent`, которое предоставляет информационную строку, содержащую подробные данные о браузере, его версии, а также операционной системы, в которой он выполняется.

Для получения местоположения посетителя вызывается метод `getCurrentPosition()`. Но следует понимать, что процесс определения местоположения занимает определенное время, в течение которого никакой уважающий себя браузер не остановит всю деятельность на странице, пока ожидает эти данные. Поэтому метод `getCurrentPosition()` исполняется асинхронно, т. е. после его запуска продолжается выполнение следующего за ним кода. Когда определение местоположения завершится, для обработки результатов активируется другой фрагмент кода.

Можно ожидать, что завершение определения местоположения сопровождается извещением, во многом подобном тому, как сообщается об окончании загрузки изображения или чтения текстового файла. Но язык JavaScript можно назвать каким угодно, но только не последовательным. И при вызове метода `getCurrentPosition()` ему передается *функция завершения* (completion function).

Далее приведен пример вызова этого метода:

```
navigator.geolocation.getCurrentPosition(
  function(position) {
    // Последний раз вас засекли здесь:
    alert("You were last spotted at (" + position.coords.latitude +
         "," + position.coords.longitude + ")");
    }
);
```

При исполнении этот код вызывает метод getCurrentPosition() и передает ему функцию. Когда браузер завершит процесс определения местоположения, он активирует эту функцию, которая выводит окно сообщения. Результаты исполнения этого кода в Internet Explorer показаны на рис. 12.2.

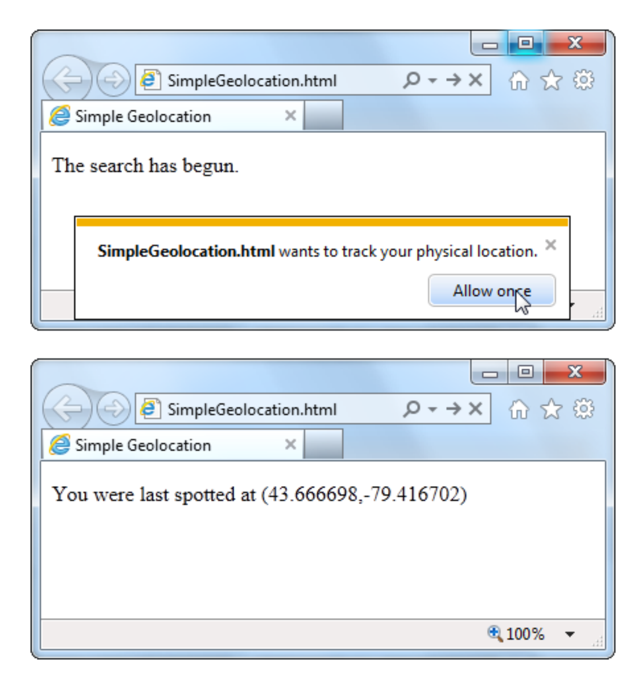


Рис. 12.2. *Вверху:* сначала нужно разрешить браузеру передать веб-серверу информацию о местоположении. *Внизу:* результаты геолокации — географические координаты посетителя

Скорее всего, чтобы не загромождать код, вместо определения функции завершения в коде вызова метода getCurrentPosition(), как сделано в предыдущем примере, ее следует определить отдельно, как показано в следующем коде:

```
function geolocationSuccess(position) {
  alert("You were last spotted at (" + position.coords.latitude +
        "," + position.coords.longitude + ")");
}
```

Потом при вызове метода getCurrentPosition() эту функцию можно передавать по ссылке:

```
navigator.geolocation.getCurrentPosition(geolocationSuccess);
```

Но не забывайте: использование геолокации допустимо только в том случае, если браузер поддерживает эту возможность и посетитель разрешит ее применение. Также разумно протестировать страницу, применяющую эту возможность, прежде чем размещать ее на веб-сервере для практического применения. В противном случае могут возникнуть проблемы (например, не будет функционировать обработка ошибок геолокации), и некоторые браузеры (скажем, Chrome) вообще не смогут определить местоположение пользователя.

Если вы спрашиваете себя: "А какая мне польза от географических координат?", мы рассмотрим ответ на этот вопрос в *разд. "Отображение карты" далее в этой главе*. Но прежде нам нужно понять, как выявлять ошибки геолокации и устанавливать некоторые параметры этой возможности.

ПРАКТИЧЕСКИЕ ЗАНЯТИЯ ДЛЯ ОПЫТНЫХ ПОЛЬЗОВАТЕЛЕЙ
Определение точности предоставленного местоположения

При успешном выполнении метода `getCurrentPosition()` код получает объект `position`, который имеет два свойства: `timestamp` (содержит время выполнения геолокации) и `coords` (содержит географические координаты).

Но свойство `coords` в свою очередь является подобъектом объекта `position` и кроме свойств `latitude` и `longitude`, определяющих географические координаты посетителя, имеет еще несколько других свойств, предоставляющих дополнительную информацию о местоположении. Это такие свойства, как `altitude` (высота над уровнем моря), `heading` (направление движения) и `speed` (скорость). Но на данный момент эти свойства не поддерживаются ни одним браузером.

Более интересным является свойство `accuracy`, которое указывает точность определенного местоположения в метрах. (Это означает, что по мере понижения точности данных местоположения значение свойства `accuracy` возрастает, что может несколько сбивать с толку.) Например, значение свойства `accuracy`, равное 2135 метрам, означает, что местоположение пользователя было определено в пределах этого расстояния.

Свойство `accuracy` полезно для определения качества результатов геолокации. Например, если значение свойства `accuracy` измеряется десятками километров, данные геолокации вряд ли имеют какую-либо практическую ценность:

```
if (position.coords.accuracy > 50000) {
  results.innerHTML =
  // Посетитель может быть где угодно на карте.
  "This guess is all over the map.";
}
```

В таком случае, возможно, разумно известить пользователя о неопределенных данных его местоположения и/или предложить ему ввести правильную информацию самому.

Обработка ошибок

Функциональность геолокации не срабатывает, если посетитель отказывается дать разрешение на ее применение. В таком случае функция завершения в нашем примере не вызывается и страница не будет знать, то ли браузер все еще пытается определить местоположение или же произошла ошибка в исполнении кода. Для решения этой проблемы при вызове метода `getCurrentPosition()` ему нужно передавать не одну функцию, а две. Первая функция, как мы уже рассмотрели, вызывается в случае успешного завершения процесса геолокации. В противном же случае вызывается вторая функция.

Далее приведен пример с использованием этих двух функций:

```
// Сохраняем элемент, в котором страница отображает результат.
var results;
```

```
window.onload  = function() {
  results = document.getElementById("results");

  // Если функциональность геолокации доступна,
  // пытаемся определить координаты посетителя.
  if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(
        geolocationSuccess, geolocationFailure
      );
    results.innerHTML = "The search has begun.";
  }
  else {
    // Данный браузер не поддерживает геолокацию.
    results.innerHTML = "This browser doesn't support geolocation.";
  }
};

function geolocationSuccess(position) {
  results.innerHTML = "You were last spotted at (" +
  position.coords.latitude + "," + position.coords.longitude + ")";
}

function geolocationFailure(positionError) {
  results.innerHTML = "Geolocation failed.";
}
```

При вызове функции ошибки геолокации браузер передает ей объект ошибки, который имеет два свойства: `code` (содержит числовой код, указывающий один из четырех типов ошибки) и `message` (содержит короткое извещение о проблеме). Обычно извещение предназначено для тестирования, а код ошибки используется кодом функции для определения, каким образом решать данную проблему.

Модифицированная функция ошибки, которая проверяет все возможные значения кода ошибки, выглядит так:

```
function geolocationFailure(positionError) {
  if (positionError.code == 1) {
    results.innerHTML =
      "Вы решили не предоставлять данные о своем местоположении, " +
      "но это не проблема. Мы больше не будем запрашивать их у вас.";
  }
  else if (positionError.code == 2) {
    results.innerHTML =
      "Проблемы с сетью или нельзя связаться со службой определения " +
      "местоположения по каким-либо другим причинам.";
  }
  else if (positionError.code == 3) {
    results.innerHTML = "Не удалось определить местоположение " +
                        "в течение установленного времени. ";
  }
```

```
  else {
    results.innerHTML =
      "Загадочная ошибка. Совершенно не понятно, что произошло.";
  }
}
```

ПРИМЕЧАНИЕ

При исполнении веб-страницы с этим кодом с локального компьютера (не с настоящего веб-сервера) функция ошибки не активируется при отказе пользователя дать разрешение на предоставление веб-серверу данных о его местоположении.

Установка параметров геолокации

Итак, мы рассмотрели, как вызывать метод `getCurrentPosition()` с двумя параметрами: функцией для обработки успеха геолокации и функцией для обработки ошибки при попытке ее выполнения. Но этому методу можно передавать еще один параметр, коим является объект, устанавливающий определенные параметры геолокации.

В настоящее время можно установить три параметра, каждый из которых соответствует отдельному свойству объекта параметров геолокации. В следующем примере устанавливается один параметр `enableHighAccuracy`:

```
navigator.geolocation.getCurrentPosition(geolocationSuccess,
geolocationFailure, {enableHighAccuracy: true});
```

А в этом примере устанавливаются все три параметра:

```
navigator.geolocation.getCurrentPosition(
   geolocationSuccess, geolocationFailure,
   {enableHighAccuracy: true,
   timeout: 10000,
   maximumAge: 60000}
);
```

В обоих этих примерах параметры геолокации устанавливаются посредством литералов объектов JavaScript. Если вы затрудняетесь разобраться со всем этим, *см. врезку "На профессиональном уровне. Что такое литералы объектов?" далее в этой главе*.

Что же означают эти свойства? Свойство `enableHighAccuracy` задействует высокоточное определение местоположения, используя систему GPS (если устройство поддерживает эту возможность и посетитель разрешил ее использование). Не устанавливайте этот параметр, если только вам не требуется получить точные координаты, т. к. ее применение сильно повышает расход заряда аккумулятора устройства браузера. По умолчанию свойству `enableHighAccuracy` присваивается значение `false`.

Свойство `timeout` определяет период времени, в течение которого страница будет ожидать получения данных геолокации, прежде чем считать попытку неудачной.

Значения `timeout` устанавливается в миллисекундах, т. е. 10 000 мс означает максимальное ожидание 10 с. Отсчет начинается *после* того, как пользователь согласится предоставить данные геолокации. По умолчанию свойству `timeout` присваивается `0`, означающий, что страница будет ожидать результаты геолокации бесконечно, не активируя ошибку тайм-аута.

Свойство `maximumAge` позволяет кэширование данных о местоположении. Например, значение `maximumAge`, равное 60 000 мс, разрешает пользоваться данными геолокации, полученными минуту назад. Это позволяет сэкономить время и вычислительные ресурсы, а также означает, что результаты геолокации будут менее точными, если посетитель перемещается. По умолчанию свойству `maximumAge` присваивается `0`, означающий, что кэшированные результаты геолокации никогда не используются. Свойству можно также присвоить специальное значение `Infinity`, в этом случае будут использоваться кэшированные данные геолокации любой давности.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Что такое литералы объектов?

В *главе 7* мы рассмотрели метод создания объектов в JavaScript, используя функцию в качестве шаблона. В частности, в начале *разд. "Отслеживание нарисованного содержимого"* мы научились использовать функцию `Circle()` для создания множества объектов кругов. А в начале *разд. "Анимация нескольких объектов"* мы использовали эту функцию для создания объектов мячиков.

Применение функции является лучшим подходом для формального определения составляющих объекта. Это позволяет получить хорошо структурированный код, что в свою очередь облегчает сложные задачи кодирования. Но иногда нужно просто быстро создать объект для одноразовой задачи. В таком случае разумно использовать *литерал объекта*, т. к. для его создания требуется всего лишь пара фигурных скобок. Чтобы создать литерал объекта, открываем фигурную скобку, перечисляем через запятую свойства, а потом закрываем фигурную скобку. Чтобы сделать код более удобочитаемым, можно использовать пробелы и переходы на новою строку, но это не обязательно. Вот пример создания литерала объекта:

```
var personObject = {
  firstName="Joe",
  lastName="Grapta"
};
```

Для каждого свойства указывается его имя и начальное значение. Таким образом, вышеприведенный код присваивает свойству `personObject.firstName` значение `Joe`, а свойству `personObject.lastName` — значение `Grapta`.

В примере данного раздела литералы объектов служат для отправки информации системе геолокации. При условии использования правильных имен свойств (т. е. названий, ожидаемых методом `getCurrentPosition()`) подход с применением литерала объекта работает идеально. Дополнительную информацию о литералах объектов, функциях объектов и обо всем другом, связанном с пользовательскими объектами JavaScript, см. здесь: **www.javascriptkit.com/javatutors/oopjs.shtml**.

Отображение карты

Определение географических координат местоположения посетителя веб-сайта — конечно же, интересный трюк. Но он быстро утрачивает свою привлекательность,

если мы не найдем этой информации какое-либо полезное применение. За этим дело не станет, т. к. в Интернете имеются целые россыпи данных географического местоположения. (Часто проблемой является преобразование этой информации в формат, полезный для веб-приложения.) Кроме того, существует несколько картографических сервисов, неоспоримым лидером которых является Google Maps. Надежные оценки обращений к этой службе дают основания полагать, что Google Maps является наиболее используемым веб-приложением для *любых* целей.

Используя Google Maps, можно создать карту любого размера любой части мира. Можно управлять взаимодействием посетителей с этой картой, генерировать маршруты для проезда из одной точки в другую и, что наиболее полезно, накладывать на эту карту свою информацию. Например, веб-страница на основе Google Maps может показывать посетителям местонахождение предприятия владельца страницы или отмечать достопримечательности экскурсии по Манхэттену. Начать ознакомление с возможностями, предоставляемыми службой Google Maps, можно с этой веб-страницы: **http://code.google.com/apis/maps/documentation/JavaScript** (или **http://code.google.com/intl/ru/apis/maps/documentation/javascript/**).

> ***Примечание***
>
> Служба Google Maps предоставляет свои услуги на бесплатной основе (даже для коммерческих сайтов) при условии бесплатного доступа к сайту. (Для платных сайтов Google предоставляет платный картографический сервис.) В настоящее время Google Maps не вставляет свою рекламу на сайты, пользующиеся ее услугами, но лицензионные условия явно оговаривают право на это в будущем.

На рис. 12.3 показана модифицированная версия страницы геолокации. Здесь полученные географические координаты пользователя служат для отображения его местоположения на карте.

Создание такой страницы не представляет никаких сложностей. Первым делом нам нужна ссылка на сценарии интерфейса API Google Maps. Эта ссылка размещается перед всеми блоками сценариев, которые используют функциональность картографии:

```
<head>
  <meta charset="utf-8">
  <title>Geolocation Map</title>
  <script src="http://maps.google.com/maps/api/js?sensor=true"></script>
      ...
</head>
```

Далее нам будет нужен элемент `<div>` для размещения динамически создаваемой карты. Для удобства обращения к этому элементу присваиваем ему однозначный идентификатор:

```
<body>
  <p id="results">Where do you live?</p>
  <div id="mapSurface"></div>
</body>
```

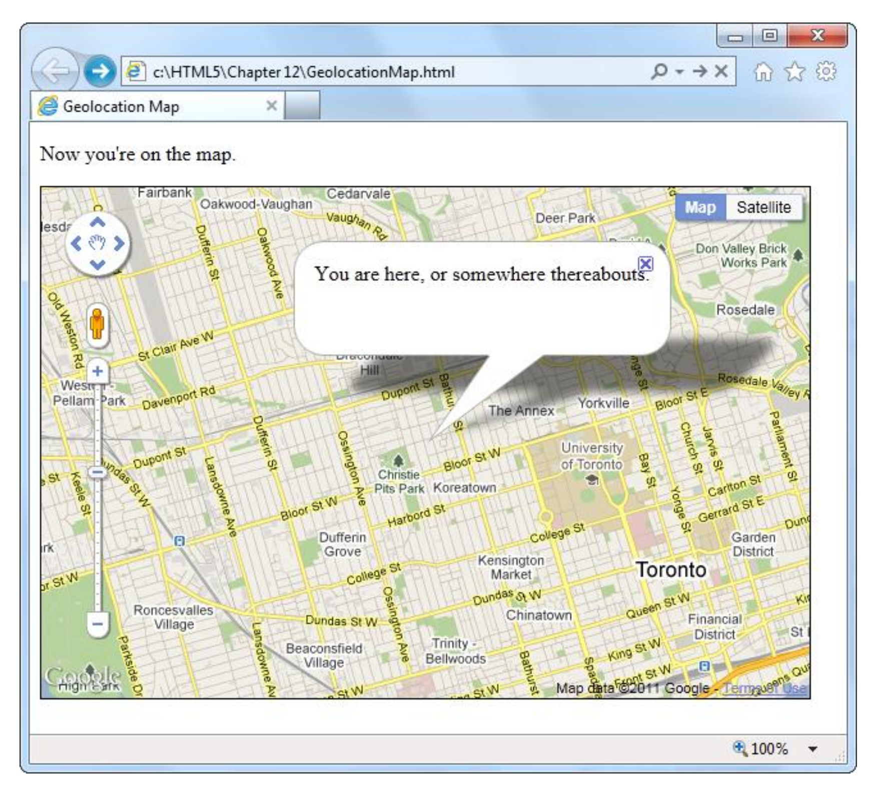


Рис. 12.3. Надпись на рисунке: *вы здесь или где-то в этом районе*. Совместно функциональность геолокации и Google Maps позволяют создавать мощные веб-приложения. Используя эти возможности, можно сгенерировать карту для любого места, применив всего лишь несколько строк кода JavaScript

Размер карты можно определить с помощью правила таблицы стилей:

```
#mapSurface {
  width:  600px;
  height: 400px;
  border: solid 1px black;
}
```

Теперь все готово к использованию функциональностей, предоставляемых Google Maps. Первым делом нужно создать поверхность карты. В этом примере карта создается при загрузке страницы, чтобы ее можно было использовать в функциях успеха или ошибки геолокации. (В конце концов, ошибка геолокации означает всего лишь то, что страница не может определить текущее местоположение пользователя. В таком случае все равно имеет смысл отображать карту, но только центрировать ее относительно точки с координатами по умолчанию.)

Далее приведен код, исполняемый при загрузке страницы. Сначала он создает карту, а потом пытается определить местоположение посетителя:

```
var results;
var map;
```

```
window.onload = function() {
  results = document.getElementById("results");

  // Устанавливаем некоторые параметры карты. В данном примере
  // устанавливаются начальный уровень масштабирования и тип карты.
  // Информацию о других параметрах см. в документации по Google Maps.
  var myOptions = {
    zoom: 13,
    mapTypeId: google.maps.MapTypeId.ROADMAP
  };

  // Создаем карту, используя установленные выше параметры.
  map = new google.maps.Map(document.getElementById("mapSurface"),
                            myOptions);

  // Пытаемся определить местоположение пользователя.
  if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(geolocationSuccess,
           geolocationFailure);
      results.innerHTML = "Поиск завершен.";
  }
  else {
    results.innerHTML = "Ваш браузер не поддерживает " +
                        "функциональность геолокации.";
    goToDefaultLocation();
  }
};
```

Совет

В документации по Google Maps (**http://code.google.com/intl/ru/apis/maps/documentation/javascript/basics.html**) показаны два способа определения местоположения посетителя — с помощью рассматриваемой в этой главе возможности геолокации или посредством резервного решения Google Gears *(см. разд. "Геолокация" ранее в этой главе)*.

Но созданная этим кодом карта еще не отображается на странице, т. к. еще не установлена географическая позиция. Для этого нужно создать специфическую глобальную точку, используя объект `LatLng`, которая потом помещается на карту посредством метода `setCenter()` карты. Далее приведен код, который выполняет все эти действия для координат посетителя:

```
function geolocationSuccess(position) {
  // Преобразуем местоположение в объект LatLng.
  location = new google.maps.LatLng(
      position.coords.latitude, position.coords.longitude);

  // Отображаем эту точку на карте.
  map.setCenter(location);
```

Этого кода будет достаточно для отображения карты наподобие показанной на рис. 12.3. Но к этой базовой карте можно добавить разные примочки, например отображение других мест во врезке или же всплывающее информационное окно. Последнее создается с помощью объекта `InfoWindow`. Далее приведен код для создания и отображения всплывающего информационного окна (см. рис. 12.3):

```
  // Создаем всплывающее информационное окно и устанавливаем
  // его текст и положение на карте.
  var infowindow = new google.maps.InfoWindow();
  infowindow.setContent("You are here, or somewhere thereabouts.");
  infowindow.setPosition(location);

  // Отображаем всплывающее окно.
  infowindow.open(map);

  results.innerHTML = "Now you're on the map.";
}
```

Наконец, в случае ошибки геолокации или отсутствия ее поддержки в браузере посетителя можно выполнить, по сути, те же действия, но используя альтернативные предустановленные координаты вместо координат посетителя:

```
function geolocationFailure(positionError) {
  ...
  goToDefaultLocation();
}

function goToDefaultLocation() {
  // Это координаты Нью-Йорка.
  var newYork = new google.maps.LatLng(40.69847, -73.95144);

  map.setCenter(newYork);
}
```

Отслеживание перемещений посетителя

Во всех рассмотренных нами до сих пор примерах мы использовали метод `getCurrentPosition()`, являющийся ключевой функциональностью геолокации. Но объект `geolocation` имеет еще два других метода, которые позволяют отслеживать перемещения посетителя и извещать страницу об этих перемещениях.

Первым из этих методов является метод `watchPosition()`, который в значительной степени похож на метод `getCurrentPosition()`. Подобно методу `getCurrentPosition()`, метод `watchPosition()` принимает параметры: функцию успеха (единственный обязательный параметр) и функцию ошибки:

```
navigator.geolocation.watchPosition(geolocationSuccess,
                                    geolocationFailure);
```

Разница между этими двумя методами состоит в том, что метод `watchPosition()` может активировать функцию успеха неоднократно — по определению местоположения в первый раз, а потом каждый раз, когда он обнаруживает новое местоположение. Для настольных компьютеров, которые никогда не перемещаются, методы `getCurrentPosition()` и `watchPosition()` имеют абсолютно одинаковый эффект.

Но в отличие от метода `getCurrentPosition()` метод `watchPosition()` возвращает число. Это число можно сохранить и передать методу `clearWatch()`, чтобы прекратить отслеживать перемещения. Этот шаг можно не выполнять и продолжать получать сообщения о перемещениях до тех пор, пока посетитель не покинет страницу:

```
var watch = navigator.geolocation.watchPosition(geolocationSuccess,
      geolocationFailure);
...
navigator.geolocation.clearWatch(watch);
```

Фоновые вычисления

В первое время после появления языка JavaScript никто особенно не беспокоился о его производительности. Этот язык был создан как простое средство для исполнения небольших сценариев в веб-странице. Можно сказать, что он был необязательной примочкой, такая себе упрощенная версия языка Java для программистов-любителей. Определенно, он не предназначался для управления каким-либо предприятием.

Но чуть меньше 20 лет спустя JavaScript завоевал Интернет. Веб-разработчики используют этот язык, чтобы добавить возможность интерактивности в веб-страницы почти любого типа, от игр и картографических инструментов до интернет-магазинов и вычурных форм.

Один из примеров недостатков языка JavaScript — это его подход к выполнению большого объема вычислений. В большинстве современных систем программирования подобные задачи выполняются незаметно в *фоновом режиме*, позволяя пользователю продолжать работать над другими аспектами задачи. Но код JavaScript всегда исполняется на *переднем плане*. Поэтому любые трудоемкие вычисления прерывают выполнение всех других задач на странице, вынуждая пользователя ожидать их завершения. Игнорирование этой проблемы выльется вам в определенное количество очень недовольных посетителей, покинувших вашу страницу с твердым намерением больше на нее не возвращаться.

Изобретательные веб-разработчики нашли несколько частичных решений данной проблемы. Эти решения основаны на разбиении долговременных задач на несколько меньших частей и исполнении этих частей по одной с помощью метода `setInternval()` или `setTimeout()`. Для некоторых типов задач это решение работает хорошо (например, это практичный способ для анимирования содержимого холста, что и было продемонстрировано в *разд. "Анимация на холсте" главы 7*). Но если

нужно выполнить одну очень долгую операцию безостановочно от начала до конца, этот метод порождает сложность и неразбериху.

Спецификация HTML5 предлагает лучшее решение в виде специализированного объекта, называющегося *веб-работником* (web worker), предназначенного для выполнения фоновых вычислений. Для выполнения долговременной задачи мы создаем нового веб-работника, даем ему необходимый код и запускаем его выполнять поставленную задачу. В процессе выполнения веб-работником своей задачи с ним можно безопасно поддерживать ограниченное взаимодействие посредством текстовых сообщений.

В табл. 12.2 показана текущая поддержка основными браузерами технологии веб-работников.

Таблица 12.2. *Поддержка веб-работников основными браузерами*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
Минимальная версия	10*	3.5	3	4	10.6	—	—

*На момент написания этой книги доступны только ранние бета-сборки этой версии.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ
Предосторожности при использовании веб-работников

Веб-работник позволяет исполнять код JavaScript в фоновом режиме, в то время как на переднем плане исполняются другие задачи. Это поднимает хорошо известную проблему современного программирования: если приложение может одновременно выполнять две задачи, одна из них может потенциально нарушить работу другой.

Проблема возникает, когда два разных фрагмента кода начинают одновременно притязать на одни и те же данные. Например, один фрагмент кода может пытаться считывать данные, в то время как другой пытается эти же данные записать. Или же оба фрагмента могут одновременно пытаться присвоить значение одной и той же переменной, в результате чего первое значение будет затерто другим. Опять же, два фрагмента кода могут пытаться манипулировать данными разными способами, тем самым устанавливая эти данные в неопределенное состояние. Возможные проблемы бесконечны, и их чрезвычайно трудно обнаружить и устранить. Часто многопотоковое приложение (т. е. приложение, которое исполняет несколько независимых *потоков* кода, коими, по сути, и являются веб-работники) работает без проблем при тестировании, но когда вы начинаете использовать его по-настоящему, в нем происходят сводящие с ума случайные ошибки.

К счастью, технология веб-работников избавлена от всех этих проблем, т. к. в ней не разрешается совместное использование данных веб-страницей и веб-работниками. Данные из веб-страницы можно *отправить* веб-работнику (или наоборот), но отправляются не исходные данные, а их копия, автоматически создаваемая JavaScript. Это означает, что просто не существует возможности, чтобы разные потоки могли бы одновременно обратиться к одной и той же ячейке памяти, создавая трудноуловимые проблемы. Конечно же, такая упрощенная модель многопотоковости также накладывает определенные ограничения на диапазон выполняемых веб-работниками задач, но это незначительное сокращение возможностей является необходимой платой за уверенность в том, что некоторые слишком амбициозные программисты не натворят дел.

ПРИМЕЧАНИЕ

В браузере Chrome попытка запустить веб-работника из локального файла закончится неудачей, если не использовать параметр `-allow-file-access-from-files`. Самый легкий способ сделать это — создать новый ярлык для запуска Chrome, который добавляет этот параметр в конец строки команды. Подробные инструкции см. здесь: **http://tinyurl.com/3j4dgcb**.

Трудоемкая задача

Для того чтобы оценить достоинства веб-работников, нам нужно найти фрагмент кода подходящей трудоемкости. Использовать веб-работников для небольших задач нет смысла. Но для вычислений, напрягающих центральный процессор, которые могут подвесить браузер больше, чем на пару секунд, веб-работники существенно меняют дело. Возьмем, например, приложение, показанное на рис. 12.4, которое генерирует простые числа в определенном диапазоне. Хотя код для решения этой задачи простой, сама задача требует трудоемких вычислений, выполнение которых может занять значительное время.

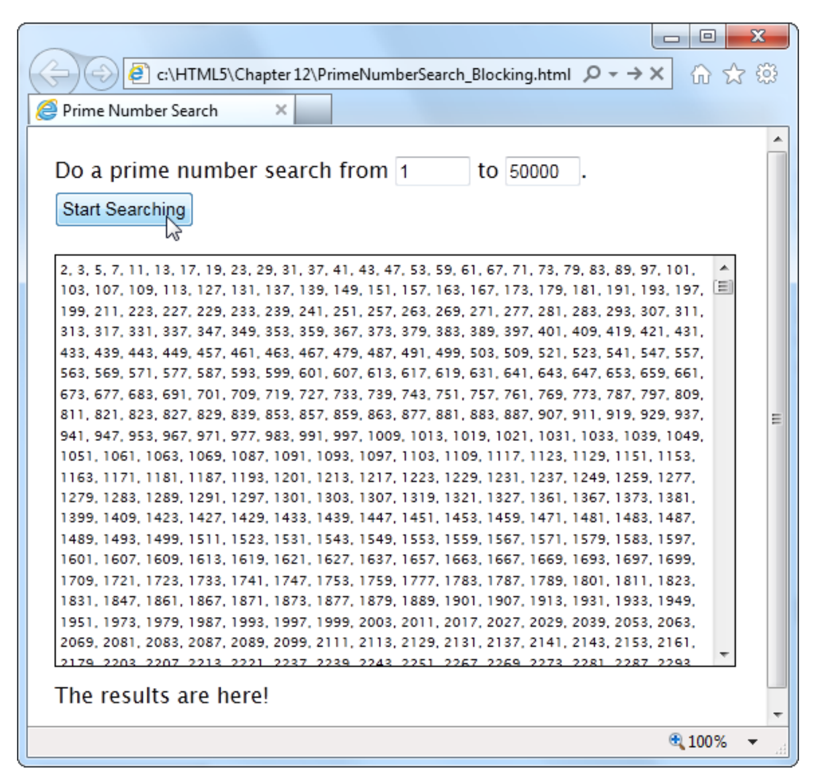


Рис. 12.4. Установите диапазон, в котором выполнять поиск простых чисел, и запустите поиск, нажав кнопку **Start Searching**. Для сравнительно узкого диапазона (как на этом рисунке — от 1 до 50 000) задача выполняется в течение нескольких секунд, не причиняя особенных неудобств пользователю. Но установите более широкий диапазон (например, от 1 до 500 000), и страница перестанет реагировать в течение нескольких минут, если не больше. Пользователь не сможет щелкать мышью элементы страницы, прокручивать ее или выполнять какие-либо другие действия; кроме этого, браузер может выдать сообщение типа "долго исполняющийся сценарий" или залить серым цветом всю страницу

Очевидно, что такую страницу можно улучшить с помощью веб-работников. Но прежде чем приступить к реализации этого улучшения, нам нужно вкратце ознакомиться с текущей разметкой и кодом JavaScript.

Эта разметка краткая и четкая. Страница использует два элемента управления `<input>`, для ввода начального и конечного числа диапазона. На ней имеется кнопка для запуска вычислений, а также два элемента `<div>` — один для вывода результатов, а другой для отображения сообщения о стоянии. Полностью вся разметка внутри элемента `<body>` выглядит так:

```
<p>Do a prime number search from <input id="from" value="1"> to
   <input id="to" value="20000">.</p>
<button id="searchButton" onclick="doSearch()">Start Searching</button>
<div id="primeContainer">
</div>
<div id = "status"></div>
```

Одним интересным аспектом является оформление элемента `<div>` для отображения списка простых чисел. Для него устанавливается фиксированная высота и ширина, а применение свойств `overflow` и `overflow-x` добавляет вертикальную полосу прокрутки (но не горизонтальную):

```
#primeContainer {
  border:        solid 1px black;
  margin-top:    20px;
  margin-bottom: 10px;
  padding:       3px;
  height:        300px;
  max-width:     500px;
  overflow:      scroll;
  overflow-x:    hidden;
  font-size:     x-small;
}
```

Код JavaScript немного длиннее, но не более сложный. Код извлекает числа из текстовых полей ввода, запускает процесс вычисления, а потом добавляет получаемые простые числа в список. Собственно математические вычисления для нахождения простых чисел этот код не выполняет. Эта задача передается отдельной функции `findPrimes()`, которая и находится в отдельном файле.

> ***Совет***
>
> Чтобы понять данный пример или принцип функционирования веб-работников, видеть все внутренности функции `findPrimes()` нет надобности. Но если вас интересует математика, положенная в основу работы этой страницы, или вы просто хотите вычислить несколько простых чисел, полный код примера можно найти на сайте этой книги по адресу **http://www.prosetech.com/html5/**.

Далее приведен полный код функции `doSearch()`:

```
function doSearch() {
  // Получаем начальное и конечное число диапазона поиска.
```

```
  var fromNumber = document.getElementById("from").value;
  var toNumber = document.getElementById("to").value;

  // Выполняем поиск простых чисел. (Это трудоемкая часть задачи.)
  var primes = findPrimes(fromNumber, toNumber);

  // Перебираем в цикле все простые числа в массиве и
  // конкатенируем их в одну длинную текстовую строку.
  var primeList = "";
  for (var i=0; i<primes.length; i++) {
    primeList += primes[i];
    if (i != primes.length-1) primeList += ", ";
  }

  // Вставляем текст с простыми числами в страницу.
  var displayList = document.getElementById("primeContainer");
  displayList.innerHTML = primeList;

  // Обновляем текст статусного сообщения, информируя пользователя
  // о происходящем.
  var statusDisplay = document.getElementById("status");
  if (primeList.length == 0) {
    // Не обнаружено ни одно простое число.
    statusDisplay.innerHTML = "Search failed to find any results.";
  }
  else {
    // А вот и результат!
    statusDisplay.innerHTML = "The results are here!";
  }
}
```

Как можно видеть, и разметка, и код краткие, простые и четкие. К сожалению, если указать большой диапазон поиска, мы также увидим, что код окажется медленным и неуклюжим, как пресловутая черепаха.

Выполнение вычислений в фоновом режиме

Возможность вычислений в фоновом режиме основана на новом объекте `Worker`. Когда нам нужно выполнить какую-либо работу в фоновом режиме, мы создаем новый объект `Worker`, снабжаем его необходимым кодом и запускаем на исполнение.

Далее приведен пример создания нового веб-работника, который исполняет код в файле PrimeWorker.js:

```
var worker = new Worker("PrimeWorker.js");
```

Исполняемый веб-работником код *всегда* находится в отдельном файле JavaScript. Такой подход препятствует неопытным, но амбициозным программистам создавать

код веб-работника, который пытается использовать глобальные переменные или обращаться к элементам страницы напрямую. С веб-работниками ни одна из таких невозможна.

Примечание

Браузеры в обязательном порядке строго разделяют код веб-страницы и код веб-работника. Например, код в файле PrimeWorker.js никаким образом не может напрямую записывать простые числа в элемент `<div>` страницы. Чтобы поместить свои данные на страницу, код веб-работника должен отправить их коду JavaScript страницы, а уже этот код отображает их.

Взаимодействие между веб-страницей и веб-работником осуществляется посредством обмена сообщениями. Чтобы отправить данные веб-работнику, вызывается его метод `postMessage()`:

```
worker.postMessage(myData);
```

Веб-работник получает событие `onMessage`, предоставляющее ему копию этих данных, и начинает их обрабатывать.

Когда веб-работнику нужно обратиться к веб-странице, он вызывает свой метод `postMessage()`, передавая ему в параметрах данные, а веб-страница получает событие `onMessage`. Этот обмен данными показан на рис. 12.5.

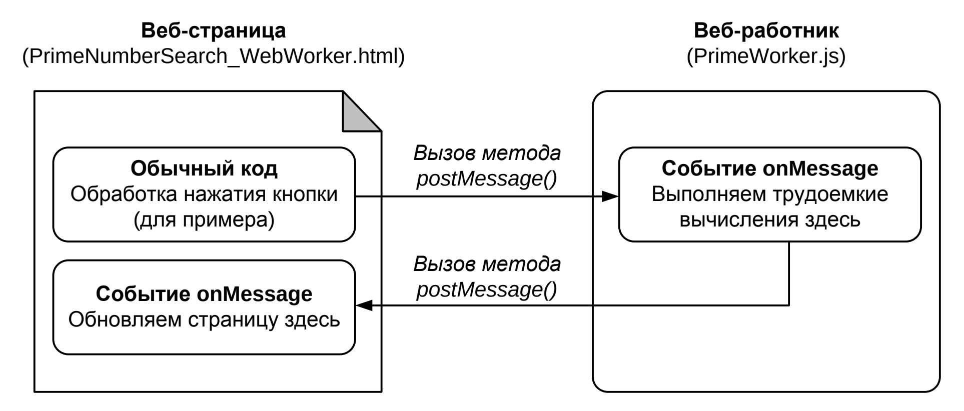


Рис. 12.5. Самый простой пример функциональности веб-работников. Процесс состоит из трех шагов: страница отправляет веб-работнику данные, веб-работник начинает вычисления, по завершению вычислений веб-работник отправляет данные на страницу

Прежде чем приступить к углубленному рассмотрению предмета веб-работников, нужно разобраться еще с одной особенностью. Функция `postMessage()` принимает только один параметр. Это представляет проблему для кода вычисления простых чисел, поскольку ему нужно передать две единицы данных — начальное и конечное значения диапазона вычислений. Проблема решается помещением этих чисел в литерал объекта *(см. врезку "На профессиональном уровне. Что такое литералы объектов?" ранее в этой главе)*. Далее приведен пример создания такого объекта с двумя свойствами (первое — `from`, для начального числа диапазона, а второе — `to`, для конечного), которым также присваиваются значения:

```
worker.postMessage(
  { from: 1,
    to:   20000 }
);
```

> **Примечание**
>
> Кстати, веб-работнику можно отправить практически любой объект. За кулисами браузер посредством технологии JSON *(см. разд. "Сохранение объектов" главы 9)* преобразует этот объект в обычный текст, дублирует его, а затем снова преобразует его в объект.

С учетом всех этих подробностей мы можем модифицировать рассмотренную ранее функцию `doSearch()`. Теперь вместо того чтобы выполнять вычисления по поиску простых чисел, функция `doSearch()` создает веб-работника и усаживает его за эту работу:

```
var worker;

function doSearch() {
  // Отключаем кнопку запуска вычислений, чтобы пользователь не мог
  // запускать несколько процессов поиска одновременно.
  searchButton.disabled = true;

  // Создаем веб-работника.
  worker = new Worker("PrimeWorker.js");

  // Подключаем функцию к событию onMessage, чтобы получать сообщения
  // от веб-работника.
  worker.onmessage = receivedWorkerMessage;

  // Получаем пределы диапазона вычислений и отправляем их веб-работнику.
  var fromNumber = document.getElementById("from").value;
  var toNumber = document.getElementById("to").value;

  worker.postMessage(
    { from: fromNumber,
        to: toNumber }
  );

  // Информируем пользователя, что вычисления выполняются.
  statusDisplay.innerHTML = "A web worker is on the job (" +
         fromNumber + " to "  + toNumber + ")  ...";
}
```

Теперь к работе приступает код в файле PrimeWorker.js. Он получает событие `onMessage`, выполняет вычисления, а затем отправляет сообщение со списком простых чисел обратно на страницу:

```
onmessage = function(event) {
  // Сохраняем в свойстве event.data отправленный веб-страницей объект.
  var fromNumber = event.data.from;
  var toNumber = event.data.to;

  // Выполняем поиск простых чисел в указанном диапазоне чисел.
  var primes = findPrimes(fromNumber, toNumber);

  // Поиск завершен. Отправляем результаты веб-странице.
  postMessage(primes);
};

function findPrimes(fromNumber, toNumber) {
  // (Собственно вычисления по поиску простых чисел выполняются
  // этой функцией.)
}
```

Когда веб-работник вызывает метод `postMessage()`, он активирует событие `onMessage`, которое в свою очередь активирует следующую функцию в веб-странице:

```
function receivedWorkerMessage(event) {
  // Получаем список простых чисел.
  var primes = event.data;

  // Отображаем список в соответствующей области страницы.
  ...

  // Разблокируем кнопку запуска поиска.
  searchButton.disabled = false;
}
```

Теперь мы можем использовать тот же код, который применяли ранее *(см. конец предыдущего раздела)*, чтобы преобразовать массив простых чисел во фрагмент текста и вставить этот текст в веб-страницу.

Хотя общая структура кода немного изменилась, его логика, по большому счету, осталась такой же. Но вот результат разительно другой. Теперь, когда выполняется поиск простых чисел, страница остается доступной. Посетитель может прокручивать ее вниз и вверх, вводить данные в текстовые поля и выбирать числа в списке предыдущего поиска. Но кроме сообщения внизу страницы, нет никаких других признаков, что где-то в фоне усиленно трудится веб-работник.

СОВЕТ

Если веб-работнику нужен доступ к коду в другом файле JavaScript, эту задачу можно с легкостью решить, используя функцию `importScripts()`. Например, функции в файле FindPrimes.js можно вызывать из кода в файле PrimeWorker.js, добавив вот эту строчку кода непосредственно перед вызовом:

```
importscripts("FindPrimes.js");
```

Обработка ошибок веб-работников

Как мы узнали, основным средством взаимодействия с веб-работниками является метод `postMessage()`. Но веб-работник может отправлять сообщения веб-странице еще одним способом — посредством события `onerror`, которое сигнализирует об ошибке:

```
worker.onerror = workerError;
```

Теперь, в случае ошибки в фоновом коде, данные об этой ошибке можно отправить странице. Следующий код веб-страницы просто отображает текст сообщения об ошибке:

```
function workerError(error)  {
  statusDisplay.innerHTML = error.message;
}
```

Но кроме свойства `message`, объект `error` также имеет свойства `lineno` и `filename`, указывающие номер строки и имя файла соответственно, в которых произошла ошибка.

Отмена исполнения фоновой задачи

Теперь, когда у нас есть базовый рабочий пример веб-работника для фоновых вычислений, мы можем улучшить его. Первым делом мы прекратим выполнение кода веб-работника.

Это можно сделать двумя способами. В первом веб-работник может остановить сам себя, вызвав метод `close()`. Но в большинстве случаев веб-работник останавливается создавшей его страницей, вызывая метод `terminate()` объекта `worker` (и это второй способ). Например, следующий код задействует кнопку прямой отмены исполнения веб-работника:

```
function cancelSearch() {
  worker.terminate();
  statusDisplay.innerHTML = "";
  searchButton.disabled = false;
}
```

Нажатие этой кнопки останавливает текущий поиск и снимает блокировку с кнопки запуска поиска. Но не забывайте, что остановив исполнение работника, вы не сможете больше отправлять ему сообщений, и его больше нельзя будет снова использовать для вычислений. Чтобы выполнить новый поиск, нужно будет создать новый объект `Worker`. (Но в этом примере это уже делается, поэтому он работает, как следует.)

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ
Резервное решение для веб-работника

К этому времени вы уже, наверное, спрашиваете себя, что следует делать, если страница с веб-работником просматривается в браузере, который не поддерживает эту возможность?

Как и в случае с геолокацией, в качестве резервного решения для веб-работников можно обратиться к проекту Google Gears, который имеет подобную возможность — *пул рабочих потоков* (worker pool). Подробную информацию см. здесь: **http://code.google.com/apis/gears/api_workerpool.html**. Но даже при использовании функциональности из проекта Google Gears все равно нужно обеспечить еще одно резервное решение для компьютеров, на которых эта система не установлена. Самое легкое и простое решение — это выполнять те же вычисления на переднем плане:

```
if (window.Worker) {
  // Функциональность веб-работников поддерживается.
  // Создаем веб-работника и усаживаем его за работу.
} else {
  // Функциональность веб-работников не поддерживается.
  // Можно просто вызвать функцию поиска простых чисел
  // и ожидать ее завершения.
}
```

При этом подходе не требуется писать никакого дополнительного кода, т. к. функция для поиска простых чисел уже существует, и ее можно вызвать напрямую, минуя вызов веб-работника. Но в случае по настоящему трудоемкой задачи этот подход может заблокировать браузер на довольно длительное время. Альтернативным (но более кропотливым) подходом будет попытка имитировать фоновые вычисления, используя методы `setInterval()` и `setTimeout()`. Например, можно создать код, который проверяет всего лишь несколько чисел за каждый интервал.

Обмен более сложными сообщениями

В последнем усовершенствовании нашего примера с веб-работником мы оснастим его возможностью предоставлять информацию о ходе выполнения задачи (рис. 12.6).

Чтобы создать индикатор хода выполнения, в процессе работы веб-работнику нужно отправлять веб-странице данные о процентах выполненного задания. Но как мы уже знаем, веб-работники имеют только один способ общаться с создавшей их страницей — посредством метода `postMessage()`. Поэтому, для того чтобы добавить индикатор хода исполнения, веб-работнику нужно отправлять два типа сообщений: сообщения о ходе исполнения (в процессе работы) и сообщение со списком простых чисел (по завершении вычислений). Кроме того, эти сообщения должны быть явно различимы.

Лучшим решением этой задачи будет добавление дополнительной информации в сообщение. Например, в сообщение о ходе исполнения веб-работник может добавить метку `"Progress"` ("Прогресс"), а в сообщение со списком простых чисел — метку `"PrimeList"` ("ПростыеЧисла").

Чтобы поместить основную информацию и метку типа сообщения в одно сообщение, нам нужно создать литерал объекта. Это точно такой же метод, как и применяемый страницей для отправки веб-работнику данных о границах диапазона вычислений. Текст метки типа сообщения присваивается в виде значения свойству `messageType`, а собственно сообщение помещается во второе свойство, `data`.

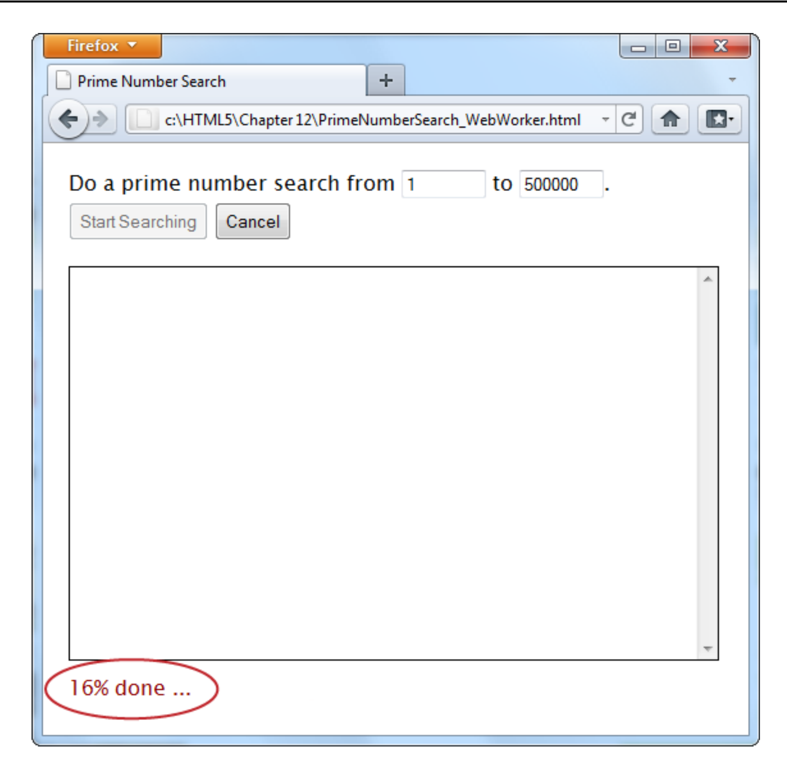


Рис. 12.6. По мере проверки, является ли число из указанного диапазона простым, код обновляет указатель хода исполнения, информируя пользователя о степени завершения задачи. Индикатор можно сделать немного повычурнее, использовав заполняемую цветом полосу (см. рис. 5.7)

Далее приведен модифицированный код веб-работника, который добавляет метку типа сообщения в сообщение со списком простых чисел:

```
onmessage = function(event) {
  // Выполняем поиск простых чисел.
  var primes = findPrimes(event.data.from, event.data.to);

  // Отправляем результаты веб-странице.
  postMessage({MessageType: "PrimeList", data: primes});
};
```

Код в функции `findPrimes()` также вызывает метод `postMessage()` для отправки сообщений веб-странице. Он использует те же свойства `messageType` и `data`. Но теперь свойство `messageType` указывает, что сообщение является сообщением о ходе исполнения, а данные содержат значение процента завершения задачи:

```
function findPrimes(fromNumber, toNumber) {
  ...
  // Вычисляем процент выполнения задачи.
  var progress = Math.round(i/list.length*100);

  // Отправляем обновленные сведения о ходе выполнения, только если
  // они изменились не менее чем на 1%.
```

```
  if (progress != previousProgress) {
      postMessage({messageType: "Progress", data: progress});
      previousProgress = progress;
  }
  ...
}
```

Когда веб-страница получает сообщение от веб-работника, она должна начать его обработку с проверки свойства `messageType`, чтобы определить тип полученного сообщения. Если сообщение содержит список простых чисел, тогда результаты выводятся в соответствующем поле страницы. Если же содержимое сообщения является извещением о ходе исполнения, тогда отображается процент завершения задачи:

```
function receivedWorkerMessage(event)  {
  var message = event.data;

  if (message.messageType == "PrimeList")  {
    var primes  = message.data;

    // Отображаем список простых чисел. Код для этого такой же,
    // как и прежде.
    ...
  }
  else if (message.messageType == "Progress")  {
    // Отображаем ход исполнения.
    statusDisplay.innerHTML = message.data + "% done ...";
  }
}
```

Примечание

Эту страницу можно оформить и по-другому. В частности, можно заставить веб-работника вызывать метод `postMessage()` всякий раз, когда он находит новое простое число. Веб-страница потом добавляет каждое новое простое число к списку и сразу же отображает его. Преимущество этого метода состоит в том, что отдельные результаты отображаются по мере их поступления. Но он также имеет и недостаток, т. к. исполнение страницы постоянно прерывается (потому что веб-работник находит простые числа довольно быстро). Идеальное решение зависит от характера задачи — сколько времени нужно на ее завершение, полезны ли частичные результаты, насколько быстро вычисляется каждый частичный результат и т. п.

ПРАКТИЧЕСКИЕ ЗАНЯТИЯ ДЛЯ ОПЫТНЫХ ПОЛЬЗОВАТЕЛЕЙ

Другое применение веб-работников

В примере поиска простых чисел веб-работники используются наиболее простым возможным способом — для выполнения одной четко определенной задачи. Страница создает нового веб-работника для каждого нового поиска. Этот веб-работник выполняет единственную задачу. Он получает одно сообщение от веб-страницы и отправляет веб-странице одно сообщение.

Но использование веб-работника не обязательно должно быть таким простым. Далее представлено несколько примеров, как можно расширить конструкцию веб-работника для выполнения более сложных задач.

- **Повторно использовать веб-работника для выполнения нескольких задач.** Веб-работник никуда не исчезает, когда он завершает работу и выполнение кода обработчика события `onMessage` доходит до конца. Он просто бездействует в режиме ожидания. Если отправить веб-работнику другое сообщение, он выходит из режима ожидания и снова приступает к работе.
- **Создать несколько веб-работников.** На веб-страницу может трудиться несколько веб-работников одновременно. Допустим, например, что мы хотим выполнять поиск простых чисел одновременно в нескольких диапазонах. Для этого можно создать нового веб-работника для каждого процесса поиска и отслеживать всех этих работников посредством массива. Когда один из веб-работников возвращает свой список простых чисел, мы добавляем этот список к странице, принимая меры для предотвращения затирания ранее записанных результатов. (Но здесь уместно предупредить, что веб-работники имеют сравнительно высокие издержки, и одновременное исполнение десятка или около того веб-работников может серьезно нагрузить компьютер.)
- **Создавать веб-работника из другого веб-работника.** Веб-работник может запускать собственных веб-работников, отправлять им сообщения и получать сообщения от них. Этот метод полезен для сложных вычислительных задач, требующих использования рекурсии, например вычисление последовательности Фибоначчи.
- **Загружать данные с помощью веб-работника.** Веб-работники могут использовать объект `XMLHttpRequest` *(см. разд. "Объект XMLHttpRequest" главы 11)*, чтобы открывать новые страницы или отправлять запросы веб-службам. Передать полученную информацию своей веб-странице они могут, вызывая метод `postMessage()`.
- **Выполнять периодические задачи с помощью веб-работника.** Веб-работники могут использовать функции `setTimeout()` и `setInterval()` точно таким же образом, как это делают обычные веб-страницы. Например, веб-работник может проверять веб-сайт на наличие новых данных каждую минуту или десять минут.

Управление историей просмотров

История просмотров — это функциональность HTML5, которая расширяет возможности объекта `history` JavaScript. Звучит просто, но нужно знать, когда и зачем следует использовать эту возможность.

Если вам никогда раньше не приходилось встречаться с объектом `history`, не стоит волноваться по этому поводу. До сих пор он не мог предложить нам ничего полезного. В действительности, традиционный объект `history` имеет только одно свойство и три основных метода. Это свойство — `length` — содержит информацию о количестве элементов в списке истории просмотров (т. е. в поддерживаемом браузером списке недавно посещенных веб-страниц). Вот пример использования этого свойства:

```
alert("You have " + history.length +
      " pages in your browser's history list.");
```

Наиболее полезным методом объекта `history` является метод `back()`. Этот метод позволяет переместить пользователя на один шаг назад в истории просмотров:

```
history.back();
```

Эффект этого метода равнозначен нажатию пользователем кнопки браузера **Назад**. Подобным образом можно использовать метод `forward()` для перемещения на один шаг вперед или метод `go()` для перехода вперед или назад на определенное количество шагов.

Но все это не представляет большой ценности, если только вы не хотите создать для веб-страницы собственные кнопки **Назад** и **Вперед**. Но HTML5 добавляет этому объекту `history` несколько дополнительных возможностей, которые можно использовать для реализации намного более амбициозных задач. Главной из этих возможностей является метод `pushState()`, который позволяет изменить URL в адресной строке браузера, не обновляя при этом содержимого страницы. Эта возможность приходится кстати в специфической ситуации, а именно при создании динамических страниц, которые незаметно загружают новое содержимое и плавно обновляют себя. В такой ситуации URL страницы и ее содержимое могут рассогласоваться. Например, если страница загрузит в динамическом режиме содержимое с другой страницы, первоначальный URL страницы не изменится, что может вызвать разнообразные проблемы с созданием закладки для этой страницы. Эту проблему можно решить с помощью отслеживания истории сеансов.

Если вы пока не видите, как это сделать — не переживайте, в следующем разделе мы рассмотрим страницу, идеально подходящую для применения истории просмотров.

Проблемы с URL

В *разд. "Получение нового содержимого" главы 11* мы рассмотрели страницу со слайд-шоу (см. рис. 11.3). Щелкая по ссылкам **Previous** и **Next**, посетитель может загружать и отображать на странице новое содержимое. Но самое примечательное в этом примере то, что каждое изображение загружается автономно, не вызывая перезагрузки всей страницы. Эта возможность достигается благодаря объекту `XMLHttpRequest`.

Но страницы с динамическим содержимым, использующие данный тип конструкции, имеют общеизвестный недостаток. В частности, хотя по загрузке нового содержимого страница изменяется, URL в строке адреса браузера остается прежним (рис. 12.7).

Теперь представьте себе, что пользователю особенно понравился один из просматриваемых слайдов, например дерево для загадывания желаний, и он сохраняет URL изображения в закладках, отправляет его по электронной почте своей приятельнице, а также делает запись об изображении с указанием URL в Твиттере. Но проблема состоит в том, что когда этот пользователь попытается возвратиться к этому изображению по сделанной закладке, или когда его приятельница щелкнет на полученной ссылке, чтобы посмотреть, что же это за особенное изображение, или когда любой из прочитавших его сообщение в Твиттере попытается перейти по указанной ссылке, все они очутятся на первом слайде. Сам пользователь, может быть, и найдет понравившееся ему изображение, но вот у остальных может не хватить терпения щелкать до пятого слайда, или они могут даже и не знать, что можно за-

гружать другие изображения. Это проблема усугубляется в случае, если слайдов огромное количество, например, поток Flickr может содержать десятки и даже сотни изображений.

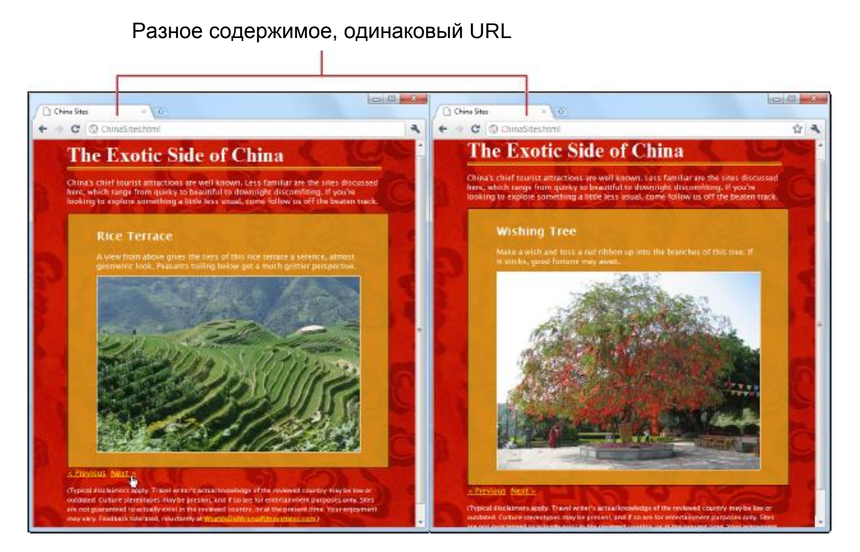


Рис. 12.7. Две версии страницы показа слайдов с разными изображениями. Несмотря на разное содержимое, обе страницы имеют одинаковый URL — ChinaSites.html

Традиционное решение: hashbang URL

Для решения этой проблемы некоторые веб-страницы добавляют дополнительную информацию в конец URL. Один из наиболее применяемых (и наиболее спорных) подходов, называющийся *hashbang*, добавляет в конец URL символы `#!`, за которыми следует идентифицирующий текст. Вот пример одного из таких URL:

```
http://jjtraveltales.com/ChinaSites.html#!/Slide5
```

Этот метод работает потому, что браузеры рассматривают все, что идет после символа `#`, как *фрагментную* часть URL. Таким образом, в случае показанного в примере URL браузер знает, что запрашивается все та же страница ChinaSites.html, но только с дополнительным фрагментом в конце.

С другой стороны, посмотрим, что произойдет, если код JavaScript изменит URL, не подставив символы `#!`:

```
http://jjtraveltales.com/ChinaSites.html/Slide5
```

Теперь браузер отправит этот запрос веб-серверу и попытается загрузить новую страницу. Но это явно не то, что нам требуется.

Как же реализовываться метод hashbang? Сначала нам нужно изменить URL, который отображается в браузере, когда страница загружает новое изображение. Это можно сделать, присвоив значение свойству `location.href` с помощью кода JavaScript. Далее, по загрузке страницы нам нужно исследовать URL, извлечь из

него фрагментную часть и получить с веб-сервера соответствующее динамическое содержимое. Все это требует довольно значительного жонглирования кодом, но использование какой-либо библиотеки JavaScript, например PathJS (**https://github.com/mtrpcic/pathjs**), может намного упростить и облегчить эту задачу.

Хотя метод hashbang получил широкое распространение, он также порождает много споров о его соответствии требованиям. Веб-разработчики начали отказываться от его использования по нескольким причинам.

- **Сложность получаемых URL.** Хорошим примером будет сайт Facebook. В прошлом, после непродолжительного просмотра этого сайта URL в адресной строке браузера засорялся дополнительной информацией, порождая таких монстров, как, например, **http://www.facebook.com/profile.php?id=1586010043#!/pages/Haskell/401573824771**. Теперь разработчики этого сайта используют историю просмотров для поддерживающих эту возможность браузеров.
- **Отсутствие гибкости.** Метод hashbang упаковывает в URL большой объем информации. Если изменить работу страницы, использующей этот метод, или ее способ сохранения информации, старые URL могут оказаться недееспособными, что вызовет крупный сбой в просмотре сайта.
- **Оптимизация поисковых систем.** Поисковые системы могут рассматривать разные URL типа hashbang, практически как один и тот же URL. В случае страницы ChineSites.html это означает, что отдельные туристические достопримечательности, представляемые конкретными слайдами, не будут проиндексированы, более того, поисковые системы могут вообще игнорировать эту информацию. Это означает, что страница ChinaSites.html не будет возвращена в результатах поиска по словам "china wishing tree". В Google пытались решить эту проблему, используя специальный синтаксис для фрагментной части URL (**http://code.google.com/web/ajaxcrawling/docs/getting-started.html**), но веб-разработчики единогласно подвергли его критике за то, что он сбивал всех с толку.
- **Cool URL важны.** Cool URL — это короткие и четкие адреса веб-страниц, которые — самое главное — никогда не меняются. С философией Cool URL в изложении создателя Интернета, Тима Бернерса-Ли (Tim Berners-Lee), можно ознакомиться на этой странице: **www.w3.org/Provider/Style/URI.html**. И независимо от того, насколько сильно ваше желание поддерживать доступность к хорошему веб-содержимому, URL типа hashbang трудно поддерживать, и они, скорее всего, не переживут следующий этап в эволюции Интернета.

Хотя веб-мастеры по-разному относятся к этому методу, большинство из них разделяет мнение, что такие адреса представляют короткий этап в развитии Интернета, и со временем им на смену придет возможность отслеживания истории сеансов.

HTML5-решение: история сеансов

HTML5 предоставляет другое решение проблемы с URL в отслеживании истории сеансов. Можно изменять URL любым образом, не требуя при этом добавления в

него странных символов, как в случае с методом hashbang. Например, когда страница ChinaSites.html загрузит пятый слайд, ее URL можно изменить так:

```
http://jjtraveltales.com/ChinaSites5.html
```

В этом случае браузер не будет пытаться запрашивать страницу ChinaSites5.html, а оставит прежнюю страницу, но загрузит для нее указанный слайд, а это нам и нужно. То же самое происходит, когда посетитель перемещается в обратном порядке в истории просмотра. Например, если посетитель перейдет к следующему слайду (и URL изменится на ChinaSites5.html), а потом возвратится назад к четвертому (возвращая URL к ChinaSites4.html), браузер сохраняет текущую страницу и активирует событие, с помощью которого можно загрузить соответствующий слайд и восстановить правильную версию страницы.

Хотя с первого взгляда все выглядит как идеальное решение, в нем есть значительный недостаток. Чтобы эта система работала должным образом, для каждого используемого URL нужно в действительности создать страницу. Для нашего примера это означает, что надо создать страницы ChinaSites1.html, ChinaSites2.html, ChinaSites3.html и т. д. Ведь посетитель может захотеть обратиться к этим страницам напрямую, например, через закладку, введя адрес вручную, щелкнув по ссылке в сообщении электронной почты и т. п. Для крупных веб-сайтов (например, Facebook или Flickr) это не представляет большой проблемы, т. к. они могут использовать серверный код и предоставить содержимое одного и того же слайда в другой упаковке. Но самостоятельному веб-разработчику для этого может потребоваться приложить несколько больше усилий. Некоторые возможные подходы к решению этой проблемы рассматриваются *во врезке "На профессиональном уровне. Создание дополнительных страниц для соответствия URL" в конце этой главы*.

Теперь, когда мы понимаем, каким образом история сеансов связана со страницами, собственно использование ее не представляет никаких трудностей. История сеансов состоит всего лишь из двух методов и одного события, добавленных к объекту `history`.

Наиболее важным является метод `pushState()`, который позволяет изменить часть URL, обозначающую страницу, в любое требуемое время. По причинам безопасности остальная часть URL изменениям не поддается. (Возможность изменять основную часть URL предоставила бы мощное средство для хакеров для подделывания веб-сайтов.)

Далее приведен пример использования метода `pushState()` для изменения конечной части URL на ChinaSites4.html:

```
history.pushState(null, null, "ChinaSites4.html");
```

Метод `pushState()` принимает три аргумента, обязательным из которых является только третий — URL, выводящийся в строке адреса браузера.

Первым параметром могут быть любые данные, сохраняемые для представления текущего состояния данной страницы. Как мы увидим далее, эти данные можно использовать, чтобы восстановить состояние страницы, если пользователь возвратится к данному URL посредством списка истории посещенных страниц браузера.

Второй параметр — это заголовок страницы, отображаемый в браузере. В настоящее время все браузеры дружно игнорируют эту подробность. Если нет надобности в установлении этих параметров, им можно просто присвоить значение `null`, как показано в примере ранее.

Далее приведен код, который нужно добавить в страницу ChinaSites.html, чтобы изменять ее URL в соответствии с текущим отображаемым слайдом:

```
function nextSlide() {
  if (slideNumber == 5) {
      slideNumber = 1;
  } else {
    slideNumber += 1;
  }

  history.pushState(slideNumber, null, "ChinaSites" +
                    slideNumber + ".html");
  goToNewSlide();
  return false;
}

function previousSlide() {
  if (slideNumber == 1)  {
      slideNumber = 5;
  } else {
    slideNumber -= 1;
  }
  history.pushState(slideNumber, null, "ChinaSites" +
                    slideNumber + ".html");
  goToNewSlide();
  return false;
}
```

Обратите внимание, что в качестве параметра состояния страницы используется номер текущего слайда. Важность этого обстоятельства станет понятной чуть позже, при рассмотрении события `onPopState`.

Результаты исполнения кода показаны на рис. 12.8.

Используя метод `pushState()`, также следует иметь в виду событие `onPopState`, которое является его естественным дополнением. В то время как метод `pushState()` вставляет новый элемент в список **История** (History) браузера, событие `onPopState` предоставляет средство для обработки этого элемента, когда посетитель возвратится к нему.

Чтобы понять работу метода, рассмотрим, что происходит, когда посетитель просматривает все слайды. В процессе просмотра URL в адресной строке браузера меняется с ChinaSites.html на ChinaSites1.html, потом на ChinaSites2.html, на ChinaSites3.html и т. д. Хотя страница в действительности не изменяется, все эти URL добавляются в историю просмотра браузера. Если пользователь щелкнет по



Рис. 12.8. При перемещении посетителя по слайдам URL изменяется, чтобы соответствовать текущему слайду. Отображаемый URL аккуратный, логичный и ссылается точно на требуемую точку в слайд-шоу

ссылке для перехода на предыдущий слайд (например, с ChinaSites3.html на ChinaSites2.html), активируется событие `onPopState`. Это событие предоставляет коду информацию состояния, сохраненную ранее посредством метода `pushState()`. Задача программиста заключается в использовании этой информации, чтобы восстановить требуемую версию страницы. В настоящем примере это означает загрузку соответствующего слайда:

```
window.onpopstate = function(e) {
  if (e.state != null) {
    // Определяем номер слайда для данного состояния.
    // (Этот номер также можно было вырезать из URL, используя
    // свойство location.href, но для этого потребуется больше работы.)
    slideNumber = e.state;

    // Запрашиваем этот слайд у веб-сервера.
    goToNewSlide();
  }
};
```

Обратите внимание, что в этом примере выполняется проверка на наличие объекта состояния, прежде чем приступать к работе. Это делается из-за того, что некоторые браузеры (включая Chrome) активируют событие `onPopState` при начальной загрузке страницы, даже если метод `pushState()` еще не вызывался.

ПРИМЕЧАНИЕ

Существует еще один метод объекта `history` — `replaceState()`, но он используется не так часто. Метод `replaceState()` можно применять для того, чтобы заменить информацию о состоянии, которая связана с текущей страницей, не добавляя при этом ничего в список **История** (History).

Поддержка браузерами истории сеансов

Функциональность отслеживания истории сеансов является сравнительно новой возможностью, хотя все последние версии основных браузеров поддерживают ее (табл. 12.3), за исключением Internet Explorer.

Таблица 12.3. *Поддержка браузерами истории сеансов*

	IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
Минимальная версия	—	4	8	5	11.5	4.2	—

Проблему, возникающую вследствие отсутствия поддержки браузером истории сеансов, можно решить несколькими способами. Если ничего не делать, просто не будут выводиться составные URL. Как раз это и происходит, если загрузить рассмотренный пример в Internet Explorer — какой бы слайд мы не загрузили, URL остается неизменно ChinaSites.html. Этот подход также применяется на сайте Flickr (для примера, откройте страницу **http://tinyurl.com/6hnvanw** в Internet Explorer).

Другой подход — активировать обновление всей страницы при загрузке нового содержимого. Этот подход имеет смысл в том случае, если предоставление качественного, значащего URL более важно, чем приятная демонстрация динамически загружаемого содержимого. Например, этот метод применяется в онлайновом хранилище кода на сайте GitHub (**http://github.com**). Но если просматривать этот сайт через браузер, поддерживающий историю сеансов, то содержимое не только загружается динамически, это делается с применением эффекта скольжения изображений.

Самый сложный вариант — использовать историю сеансов там, где это возможно, а где невозможно, применять резервное решение в виде URL типа hashbang. (Этот метод используется на Facebook.) Недостатком данного метода является необходимость применять два разных подхода в одной и той же странице. Можно также использовать заплатку JavaScript, которую можно загрузить здесь: **http://github.com/balupton/history.js**.

НА ПРОФЕССИОНАЛЬНОМ УРОВНЕ

Создание дополнительных страниц для соответствия URL

Технология истории сеансов придерживается первоначальной философии Интернета: каждый фрагмент содержимого должен иметь свой однозначный, стойкий URL. К сожалению, это означает, что необходимо обеспечить, чтобы эти URL позволяли посетителям возвратиться к понравившемуся им содержимому. И это намного более трудная задача. Например, если посетитель запрашивает страницу ChinaSites3.html, вам необходимо взять основное содержимое из ChinaSites.html и содержимое слайда из ChinaSites3_slide.html и каким-то образом объединить.

Опытный программист может написать серверный сценарий, который перехватывает этот запрос и на лету выполняет сборку конечного содержимого из частей. Но если у вас нет хороших навыков программирования, вам нужно использовать другой подход.

Самым простым решением будет создать отдельный файл для каждого URL, иными словами, в действительности создать файлы ChinaSites1.html, ChinaSites2.html, ChinaSites3.html и т. д. Конечно же, не следует дублировать содержимое слайда в нескольких местах (например, как в ChinaSites3.html, так и в ChinaSites3_slide.html), т. к. это создаст большую проблему с сопровождением таких страниц. К счастью, существуют два простых метода, которые могут упростить эту задачу.

* **Использование серверных вставок.** Если ваш веб-сервер поддерживает этот метод (как делает большинство серверов), можно использовать специальную инструкцию:

  ```
  <!--#include file="footer.html" -->
  ```

  Хотя эта строка выглядит, как будто комментарий, в действительности это инструкция веб-серверу открыть указанный файл и вставить его содержимое в этом месте разметки. Посредством этого метода можно вставить основное содержимое и содержимое слайда в страницу для каждого отдельного слайда. При этом чтобы создать основную оболочку страницы, потребуется всего лишь несколько строчек разметки в файле веб-страницы для каждого конкретного слайда (ChinaSites1.html, ChinaSites2.html и т. д.).

* **Использование шаблонов средства веб-разработки.** Такие средства веб-разработки, как Adobe Dreamweaver и Microsoft Expression Web, позволяют формировать шаблоны страниц, на основе которых можно создавать любое количество страниц со всеми исходными подробностями. Таким образом, создав шаблон с основным содержимым и форматированием, на его основе можно легко и быстро построить все страницы для отдельных слайдов.

IV

ЧАСТЬ IV

Приложения

1

ПРИЛОЖЕНИЕ 1

Очень краткое введение в CSS

Мы нисколько не преувеличим, если скажем, что современный веб-дизайн был бы невозможен без стандарта CSS (Cascading Style Sheets, каскадные таблицы стилей). Этот стандарт позволяет передать задачи форматирования любой сложности в распоряжение отдельному документу — *таблице стилей* (style sheet). Это делает разметку веб-страницы аккуратной, ясной и читаемой.

Чтобы извлечь наибольшую пользу из стандарта HTML5 (и этой книги), нужно знать CSS. Если у вас хороший уровень знаний в этой области, можно пропустить данное приложение и продолжать изучать остальной материал в этой книге, обращая особое внимание на *главу 8*, в которой представляются новые возможности форматирования, добавленные в CSS3. Но если ваши навыки CSS несколько заржавели, это приложение поможет вам освежить их, прежде чем продолжать изучение дальнейшего материала.

ПРИМЕЧАНИЕ

Это приложение содержит очень краткое, далеко не всеохватывающее, освещение стандарта CSS. Если после ознакомления с содержащимся в нем материалом вы все еще чувствуете себе неуверенным в этой области, прочитайте какую-либо книгу, в которой предмет CSS рассматривается более подробно.

Добавление стилей в веб-страницу

Существуют три способа добавления стилей на страницу.

Первый — вставить информацию о стиле непосредственно в требующий форматирования элемент разметки, используя атрибут `style`. Вот пример такого типа форматирования, которое изменяет цвет заголовка:

```
<h1 style="color: green">Inline Styles are Sloppy Styles</h1>
```

В то время как этот подход удобен, он ужасно засоряет разметку.

Вторым подходом будет вставить всю таблицу стилей в элемент `<style>`, который помещается в блок `<head>` страницы:

```
<head>
  <title>Embedded Style Sheet Test</title>
  <style>
  ...
  </style>
</head>
```

Здесь форматирование страницы отделено от ее разметки, но оба эти компонента продолжают находиться в одном файле. Этот подход можно применять для одноразовых задач форматирования (когда нет надобности использовать эти параметры для форматирования других страниц). Но страницы с применением этого метода получаются большого объема, вследствие чего он не очень подходит для настоящих, профессиональных веб-сайтов.

А третий способ — это разместить форматирование в отдельном файле и связать его с форматируемой страницей, вставив элемент `<link>` в блок `<head>` страницы. Вот пример кода, дающего указание браузеру применять стили из файла SampleStyles.css:

```
<head>
  <title>External Style Sheet Test</title>
  <link rel="stylesheet" href="SampleStyles.css">
</head>
```

Это наиболее часто применяемый метод форматирования и при этом наиболее действенный. Определенные таким образом стили можно использовать для форматирования других страниц. При желании стили можно распределить между несколькими файлами, а страницу HTML связать с любым количеством файлов таблиц стилей.

Примечание

Современный веб-дизайн зиждется на простой философии: HTML-разметка используется для структурирования страницы в логические блоки (например, разделы, заголовки, списки, изображения и ссылки), а таблицы стилей CSS применяются для форматирования страницы (указывая шрифты, цвета, поля, фон и позиционирование содержимого). Следуйте этому правилу, и ваши страницы будут компактными и четкими, легко поддающимися редактированию. Использование таблиц стилей также позволяет изменить форматирование всего веб-сайта простой модификацией связанной с ним таблицы стилей. (В высшей степени впечатляющую демонстрацию магии таблиц стилей можно увидеть на сайте **www.csszengarden.com**, которому можно придать более чем 200 разных обликов, просто подключая другие таблицы стилей.)

Анатомия таблицы стилей

Таблица стилей — это текстовый файл, который помещается на веб-сервер вместе со страницами HTML. Файл содержит одно или несколько *правил*, порядок следования которых в файле не имеет значения.

Каждое правило определяет один или несколько аспектов форматирования одного или нескольких элементов HTML. Далее приводится структура простого правила стиля:

```
селектор {
      свойство: значение;
      свойство: значение;
}
```

Значения элементов правила следующие.

- **Селектор** определяет тип содержимого, которое нужно форматировать. Браузер выискивает в странице все совпадающие с селектором элементы и применяет к ним указанное для данного селектора форматирование. Селектор можно указывать многими разными способами, но одним из самых простых подходов (демонстрируемый в следующем примере) будет определение элементов, подлежащих форматированию посредством названия этих элементов. Например, можно создать селектор, который выбирает в странице все заголовки первого уровня `<h1>`.
- **Свойство** определяет составляющую элемента, которую нужно форматировать. Это может быть цвет, шрифт, выравнивание содержимого ячейки и т. п. Правило может содержать любое количество свойств.
- **Значение** определяет значение свойства. Например, если есть свойство `color`, его значение может быть `light blue` (светло-синий) или `green` (зеленый).

Вот пример правила для форматирования заголовка первого уровня:

```
h1 {
      text-align: center;
      color: green;
}
```

Скопируйте это правило в текстовый файл и сохраните его с расширением css (например, SampleStyles.css). Потом создайте простую веб-страницу, содержащую, по крайней мере, один заголовок первого уровня, и вставьте в нее элемент `<link>` со ссылкой на этот файл таблицы стилей. Наконец, откройте эту страницу в браузере. Вы увидите, что все заголовки первого уровня будут зеленого цвета и выровнены по центру.

Свойства CSS

В предыдущем примере правило таблицы стилей содержит два свойства: `text-align` (горизонтальное выравнивание текста) и `color` (цвет текста).

Кроме этих двух, существует множество других свойств форматирования, наиболее употребляемые из которых приведены в табл. П1.1. Кстати, эта таблица содержит почти все свойства стилей, которые применяются в примерах этой книги (за исключением новых свойств CSS3, которые рассматриваются в *главе 8*).

> **Совет**
>
> Если у вас нет под рукой справочника по таблице стилей, для просмотра всех свойств, перечисленных в этой таблице (и многих других), воспользуйтесь сайтом **www.htmldog.com/reference/cssproperties**. Здесь же можно узнать более подробную информацию о каждом свойстве, включая краткое описание его действия и разрешенные значения.

Таблица П1.1. *Наиболее используемые свойства таблиц стилей*

Свойство	**Название**
Цвет	`color`, `background-color`
Интервал	`margin`, `padding`, `margin-left`, `margin-right`, `margin-top`, `margin-bottom`
Границы	`border-width`, `border-style`, `border-color`, `border` (для установки ширины, стиля и цвета границ за один прием)
Выравнивание текста	`text-align`, `text-indent`, `word-spacing`, `letter-spacing`, `line-height`, `white-space`
Шрифты	`font-family`, `font-size`, `font-weight`, `font-style`, `font-variant`, `text-decoration`, `@font-face` (для использования вычурных шрифтов; *см. разд. "Типография для Интернета" главы 8*)
Размер	`width`, `height`
Позиция	`position`, `left`, `right`, `float`, `center`
Графика	`background-image`, `background-repeat`, `background-position`

Форматирование элементов посредством классов

Приведенное ранее в качестве примера правило таблицы стилей форматирует в документе все заголовки первого уровня. Но в более сложных документах может потребоваться отформатировать только один элемент или разные группы элементов одного типа по-разному.

Для решения этой задачи элементу присваивается название посредством атрибута `class`. Далее приведен пример форматирования с использованием класса элемента:

```
<h1 class="ArticleTitle">HTML5 is Winning</h1>
```

Теперь можно создать правило таблицы стилей, которое форматирует только этот заголовок. В данном случае название селектора начинается с точки, после которой следует имя класса:

```
.ArticleTitle {
  font-family: Garamond, serif;
  font-size: 40px;
}
```

Теперь все элементы, которым посредством атрибута `class` присвоено имя `ArticleTitle`, будут отформатированы стилем, указанным в определении данного класса в таблице стилей.

Атрибут `class` можно использовать с любым количеством элементов. В действительности, это и есть идея в основе использования классов. Типичная таблица стилей наполнена правилами классов, опрятно разбивающих разметку веб-страницы на блоки, к которым можно применять стили.

Наконец, следует отметить, что можно создать селектор, использующий тип элемента и имя класса, как показано в следующем примере:

```
h1.ArticleTitle {
  font-size: 40px;
}
```

Данный селектор создает класс `ArticleTitle`, который применим только для элементов `<h1>`. Иногда этот тип правила стилей служит просто для ясности. Например, разработчик хочет четко указать, что класс `ArticleTitle` применяется только для форматирования заголовков первого уровня, и не должен применяться для форматирования других элементов. Но в большинстве случаев веб-дизайнеры создают простые классы, без привязки к определенным элементам.

> **Примечание**
>
> Разные селекторы могут перекрывать друг друга. Если к одному и тому же элементу применяется несколько селекторов, используются стили, указанные в обоих этих селекторах, при этом наиболее общие применяются первыми. Если, например, имеется правило для всех заголовков и правило для класса `ArticleTitle`, первым применяется правило для всех заголовков, а потом правило класса. В результате правило класса может затереть свойства, установленные правилом для всех заголовков. Если оба правила одинаково специфичны, выигрывает то, которое в таблице стилей определено последним.

Комментарии в таблицах стилей

В сложной таблице стилей иногда стоит делать заметки, чтобы напоминать себе (или дать знать другим) о причине создания правила и выполняемом им форматировании. Подобно HTML, CSS позволяет добавлять комментарии в таблицы стилей, которые игнорируются браузерами. Но комментарии CSS отличаются от комментариев HTML. Они всегда начинаются с символов `/*`, а заканчиваются символами `*/`. Вот пример комментария CSS:

```
/* Заголовок основной статьи страницы. */
.ArticleTitle {
  font-size: 40px;
}
```

Продвинутые таблицы стилей

Мы рассмотрим пример таблицы стилей немного позднее, после того, как познакомимся с некоторыми подробностями создания таблиц стилей.

Структурирование страницы с помощью элементов <*div*>

При работе с таблицами стилей часто приходится использовать элемент `<div>` в качестве контейнера для содержимого:

```
<div>
  <p>Here are two paragraphs of content.</p>
  <p>In a div container.</p>
</div>
```

Сам по себе элемент `<div>` ничего не делает, но он предоставляет удобное место для применения форматирования таблиц стилей на основе классов. Далее приведено несколько примеров.

- **Унаследованные значения.** Некоторые свойства CSS являются наследуемыми, т. е. значение, присвоенное одному элементу, автоматически присваивается всем элементам внутри этого элемента. Одним из примеров будет правило с набором свойств шрифта. Если применить такое правило к элементу `<div>`, весь текст внутри этого элемента получит данное форматирование (кроме элементов текста, к которым могут быть применены отдельные правила форматирования).
- **Контейнеры.** Элемент `<div>` — это естественный контейнер. Добавим к нему границы, интервалы и заливку фона (или изображение вместо заливки), и мы получим способ для выделения выбранного содержимого.
- **Колонки.** Содержимое профессиональных веб-сайтов часто разбивается на две или три колонки. Один из способов осуществления этого — поместить содержимое каждой колонки в элемент `<div>`, а потом разместить каждый из этих элементов в требуемом месте с помощью свойств CSS для позиционирования.

Совет

Теперь, когда спецификацией HTML5 были введены семантические элементы, элемент `<div>` не играет такой центральной роли, как раньше. Если его можно заменить другим, более значащим семантическим элементом (например, `<header>` или `<figure>`), следует делать такую замену. Но когда не подходит никакой другой элемент, элемент `<div>` остается универсальным средством. Подробное описание всех новых семантических элементов *см. в главе 2*.

Элемент `<div>` имеет младшего брата — `<span>`. Подобно элементу `<div>`, элемент `<span>` не имеет встроенного форматирования. Разница между этими двумя элементами состоит в том, что элемент `<div>` служит контейнером для абзацев или блоков содержимого, а элемент `<span>` является строчным и служит контейнером для фрагментов содержимого меньшего размера внутри блочных элементов. Например, элемент `<span>` можно использовать для форматирования нескольких слов внутри абзаца.

Примечание

Использование CSS способствует качественному дизайну. Каким образом? Чтобы использовать CSS эффективно, нужно должным образом спланировать структуру веб-страницы. Таким образом, необходимость использования CSS вынуждает даже непрофессиональных веб-разработчиков подходить серьезно к организации содержимого своих веб-страниц.

Множественные селекторы

Правило таблицы стилей можно определить для нескольких элементов или классов. Для этого селекторы разделяются запятыми.

Возьмем, например, следующие два разных уровня заголовка, которые имеют одинаковый шрифт, но разного размера:

```
h1 {
  font-family: Impact, Charcoal, sans-serif;
  font-size: 40px;
}
h2 {
  font-family: Impact, Charcoal, sans-serif;
  font-size: 20px;
}
```

Параметр `font-family` можно выделить в отдельное правило, которое применяется к заголовкам обоих уровней, а размер шрифта по-прежнему определяется отдельным правилом для каждого уровня:

```
h1, h2 {  font-family: Impact, Charcoal, sans-serif;  }
h1 {  font-size: 40px;  }
h2 {  font-size: 20px;  }
```

В этом отношении важно понимать, что такой дизайн не обязательно лучше. Часто разумнее дублировать параметры, т. к. это допускает наибольшую гибкость для изменения форматирования в будущем, если это понадобится. Слишком много общих свойств усложняет модифицирование одного типа элемента или класса, не затрагивая при этом другого.

Контекстные селекторы

Контекстный селектор применяет правило к элементу, расположенному *внутри* другого элемента. Далее приведен пример контекстного селектора:

```
.Content  h2 {
  color:  #24486C;
  font-size:  medium;
}
```

Этот селектор выбирает в разметке элемент класса `Content`, а внутри этого элемента — элемент `<h2>` и применяет к нему указанное в правиле форматирование. Вот пример элемента, который выбирается данным селектором:

```
<div class="Content">
  ...
  <h2>Mayan Doomsday</h2>
  ...
</div>
```

В этом примере контекстного селектора первый селектор является классом, а второй — типом элемента. Но можно использовать любую комбинацию селекторов класса и типов элемента. Например, далее приведен пример контекстного селектора для комбинации классов:

```
.Content .LeadIn {
  font-variant:  small-caps;
}
```

Этот селектор выбирает в разметке элемент класса `Content`, а внутри этого элемента — элемент класса `LeadIn`. Далее приведен пример элемента, который выбирается данным селектором:

```
<div class="Content">
  <p><span class="LeadIn">Right now</span>, you're probably feeling
      pretty good. After all, life in the developed world is
      comfortable ...</p>
</div>
```

Немного попрактиковавшись с контекстными селекторами, вы увидите, что они довольно простые и очень полезные.

Идентификаторы

Селекторы классов имеют близких родственников — *идентификаторы* (или ID-селекторами). Подобно селектору класса, идентификатор позволяет применять форматирование только к выбранным элементам. Также подобно селектору класса, идентификатору можно присвоить описательное имя. Но вместо точки название идентификатора начинается с символа `#`:

```
#Menu {
  border-width:  2px;
  border-style:  solid;
}
```

Как и с правилами классов, браузеры используют правила идентификаторов только при условии их явного указания в разметке, для чего предназначен атрибут `id` элемента. Например, следующий код применяет определенное ранее правило `#Menu` к элементу `<div>`:

```
<div id="Menu">...</div>
```

Сейчас вы, наверное, задаете себе вопрос: какая же разница между селекторами класса и идентификаторами, т. к. последний кажется в точности таким же, как и первый? Разница следующая: данный идентификатор можно присвоить только *одному* элементу на странице. Для нашего примера это означает, что только одному элементу `<div>` можно присвоить идентификатор `Menu`. Это ограничение не распространяется на имена классов, которые можно применять с неограниченным числом элементов.

Таким образом, идентификатор разумно выбирать, если нужно отформатировать один никогда не повторяющийся элемент на странице. Преимущество использования для этого идентификатора вместо селектора класса состоит в том, что первый явно указывает особую важность данного элемента. Например, если в разметке имеется идентификатор `Menu` или `NavigationBar`, разработчик знает, что данная страница имеет лишь одно меню или навигационную панель. Конечно же, применение идентификаторов вовсе не является обязательным. Некоторые веб-разработчики используют селекторы класса для всего подряд, независимо, является блок уникальным или нет. По сути, это всего лишь вопрос личных предпочтений.

ПРИМЕЧАНИЕ

Атрибут `id` также играет важную роль в JavaScript, позволяя разработчику выделить конкретный элемент для манипулирования им посредством кода. В примерах в этой книге применяются правила идентификаторов во всех случаях, когда элемент уже имеет атрибут `id` для JavaScript. Это позволяет избежать одновременного использования обоих атрибутов — `id` и `class`. Во всех других случаях применяются правила класса независимо от того, является элемент единственным или нет.

Селекторы псевдоклассов

До сих пор все рассматриваемые нами селекторы были простыми. В них для выбора элемента учитывается один явный тип информации, например тип элемента, название класса или идентификатора. Селекторы псевдоклассов несколько посложнее. В них принимается во внимание дополнительная информация, такая, которая может и не указываться в разметке или, возможно, основана на действиях пользователя.

На протяжении существования CSS браузеры поддерживали всего лишь несколько псевдоклассов, которые в основном предназначались для форматирования ссылок. Псевдокласс `:link` форматирует все непосещенные ссылки. Псевдокласс `:visited` — все посещенные ссылки. Псевдокласс `:hover` форматирует ссылку, на которую пользователь навел указатель мыши, а псевдокласс `:active` форматирует ссылку, по которой пользователь щелкнул, но еще не отпустил кнопку мыши. Как можно видеть, названия псевдоклассов всегда начинаются с двоеточия.

Для практики рассмотрим правило таблицы стилей, которое использует псевдоклассы для создания страницы и вводит посетителя в заблуждение — посещенные ссылки отображаются синим цветом, а непосещенные — красным (т. е. противоположно тому, как это делается в действительности):

```
a:link     { color: red; }
a:visited  { color: blue; }
```

С псевдоклассами также можно использовать имена классов:

```
.BackwardLink:link {
  color:   red;
}
.BackwardLink:visited {
  color: blue;
}
```

Теперь элемент привязки должен указать имя класса, чтобы отображать новый стиль:

```
<a class="BackwardLink" href="...">...</a>
```

Псевдоклассы служат не только для форматирования ссылок. Так, псевдокласс `:hover` полезен для эффектов анимации и создания вычурных кнопок. Этот псевдокласс используется с возможностями перехода *(см. разд. "Создание эффектов перехода" главы 8)*.

ПРИМЕЧАНИЕ

Спецификация CSS3 вводит несколько более продвинутых псевдоклассов, которые принимают во внимание иные подробности, например позицию элемента относительно других элементов или состояние элемента управления `input` в веб-форме. Эти псевдоклассы не рассматриваются в данной книге, но вы можете узнать о них больше в статье журнала "Smashing Magazine" на сайте **http://tinyurl.com/3p28wau**[1].

Селекторы атрибутов

Спецификация CSS3 определяет новую возможность выбора атрибутов, позволяющую форматировать конкретный тип элемента, одному из атрибутов которого присвоено специфическое значение. Рассмотрим, например, правило, которое применимо только к текстовым полям:

```
input[type="text"]  {
  background-color:silver;
}
```

Сначала этот селектор выбирает все элементы `<input>`, а потом среди этих элементов выбирает и форматирует только те, у которых значение атрибута `type` равно `text`. В отношении следующей разметки это означает, что первый элемент `<input>` получит фон серебристого цвета, но второй — нет:

```
<label for="name">Name:</label><input id="name" type="text"><br>
<input type="submit" value="OK">
```

Указывать значение атрибута `type="text"` в первом элементе `<input>` необязательно, т. к. это значение по умолчанию. Если не указать это значение, селектор атрибута будет все равно работать, т. к. он руководствуется текущим значением атрибута, и ему безразлично, как это значение установлено в разметке.

Аналогично можно создать правило, которое форматирует надпись для этого текстового поля, но игнорирует все другие надписи:

```
label[for="name"]  {
  width: 200px;
}
```

ПРИМЕЧАНИЕ

Селекторы атрибутов можно применять и более изобретательно. Например, можно составить правило для выбора элемента по комбинации значений атрибутов или же по части значения атрибута. Такие методы оригинальны, но слишком усложняют обычную таблицу стилей. Подробную информацию см. в стандарте CSS3 для селекторов по адресу **www.w3.org/TR/css3-selectors/#selectors**.

---

[1] Вообще даже в русскоязычном сегменте Интернета можно найти массу статей по этому поводу. Достаточно в поисковой строке любой поисковой системы набрать запрос "как использовать псевдоклассы CSS3" или "псевдоклассы CSS3", и вашему вниманию будет предложен длинный список найденных ссылок. — *Ред.*

Экскурсия по таблице стилей

В *главе 2* рассматривается использование новых семантических элементов HTML5, модифицирующих простую, но аккуратно отформатированную страницу ApocalypsePage_Original.html (рис. П1.1).

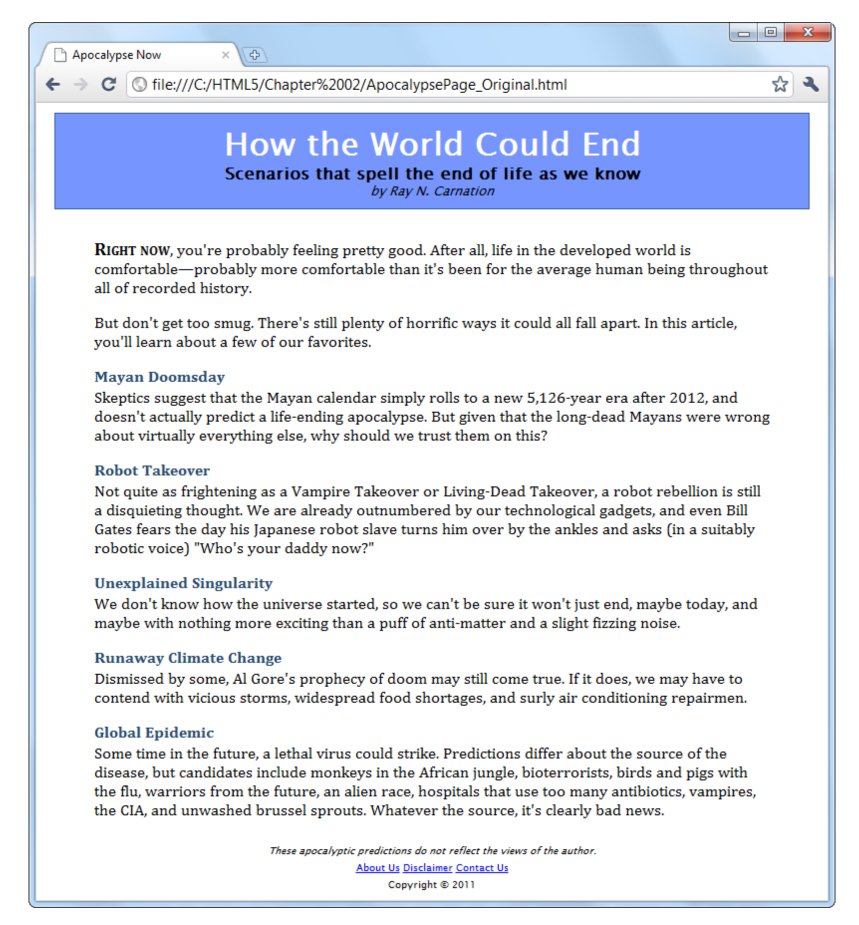


Рис. П1.1. На этой странице используются простые стили, но они следуют основным организационным принципам, которые применяются на практике при изложении материала этой книги

Эта страница связана с таблицей стилей ApocalypsePage_Original.css:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Apocalypse Now</title>
  <link rel="stylesheet" href="ApocalypsePage_Original.css">
</head>
...
```

Это простая и сравнительно краткая таблица стилей, насчитывающая 50 с лишним строк кода. В этом разделе мы рассмотрим каждое из содержащихся в ней правил.

Таблица стилей начинается с правила, нацеленного на элемент <body>, который является корневым элементом всей веб-страницы. Это лучшее место для определения наследуемых значений, которые нужно применять по умолчанию ко всему документу, т. е. к полям, отбивкам, цвету фона, шрифтам и ширине:

```
body {
  font-family: "Lucida Sans Unicode", "Lucida Grande", Geneva, sans-serif;
  max-width: 800px;
}
```

Устанавливая в CSS свойство font-family, нужно следовать двум правилам. Первое: шрифт должен быть пригодным для веб-страниц — один из немногих шрифтов, о которых заведомо известно, что они работают практически на всех подключенных к Интернету компьютерах (список основных шрифтов см. на сайте **http://www.fonttester.com/help/list_of_web_safe_fonts.html**, а другие, более рискованные шрифты, см. на сайте **http://www.speaking-in-styles.com/web-typography/Web-Safe-Fonts**). Второе: список шрифтов должен начинаться с конкретного варианта шрифта, за которым следуют возможные резервные шрифты. В конце списка указывается инструкция serif или sans-serif (две инструкции для шрифтов, которые поддерживаются всеми браузерами). Если вы хотите использовать вычурный шрифт, который посетитель должен загрузить с вашего веб-сервера, ознакомьтесь с возможностью CSS3 для внедрения шрифтов, рассматриваемой в *разд. "Типография для Интернета" главы 8*.

Правило для элемента <body> также устанавливает максимальную ширину в 800 пикселов. Таким способом предотвращается использование длинных, трудночитаемых строк текста, когда окно браузера разворачивается на всю ширину экрана. Эту проблему также можно решить другими методами, включая разбиение текста на колонки *(см. разд. "Размещение текста в несколько колонок" главы 8)*, использование запросов о возможностях *(см. разд. "Запросы о возможностях отображения" главы 8)* или создание дополнительных колонок, чтобы заполнить свободное пространство. Но хотя установка фиксированной ширины в 800 пикселов — не самое привлекательное решение, оно является наиболее распространенным.

Далее определяется правило для класса, которое форматирует область верхнего колонтитула страницы:

```
.Header {
  background-color: #7695FE;
  border:     thin #336699 solid;
  padding:    10px;
  margin:     10px;
  text-align: center;
}
```

Примечание

В первоначальном примере (использованном здесь) для размещения колонтитула применяется простой элемент <div>, которому присвоено имя класса Header. Но в *главе 2* рассматривается возможность использования вместо этого элемента нового HTML5-элемента <header>.

Это правило содержит большой объем информации. Свойство background-color можно установить, подобно всем цветам CSS, используя название цвета (что дает сравнительно небольшой выбор цветов), код HTML-цвета (как сделано в этом примере) или функцию rgb() (в параметрах которой указываются красная, зеленая и синяя составляющие цвета). В примерах этой книги применяются все три подхода, при этом метод с названиями цветов используется в простых примерах, а функция rgb() в более сложных.

Кстати, все HTML-цвета можно представить с помощью функции rgb() и наоборот. Например, цвет в предыдущем примере можно указать посредством функции rgb() так:

```
background-color: rgb(118,149,254);
```

Совет

Получить значения RGB для требуемого цвета можно с помощью онлайнового цветоподборщика или же посредством какой-либо графической программы, хотя бы Paint.

В правиле для колонтитула также определяется тонкая граница вдоль его края. Для этого используется "все в одном" свойство border, посредством которого устанавливается толщина и цвет границ, а также их стиль (например, solid — сплошная, dashed — штриховая, dotted — пунктирная, double — двойная, groove — с выемкой, ridge — рельефная, inset или outset — псевдотрехмерная).

После фона и границ устанавливается отбивка (padding) в 10 пикселов (расстояние между содержимым блока верхнего колонтитула и его границами), а потом поля (margin) в 10 пикселов (расстояние между границами колонтитула и границами веб-страницы). Наконец, свойству text-align присваивается значение, центрирующее по горизонтали текст внутри колонтитула.

В представленных далее трех правилах используются контекстные селекторы для форматирования элементов внутри верхнего колонтитула. Следующее правило форматирует заголовки первого уровня (элементы <h1>) внутри верхнего колонтитула:

```
.Header h1 {
  margin:    0px;
  color:     white;
  font-size: xx-large;
}
```

Совет

Размер шрифта можно указывать посредством ключевых слов (как значение xx-large в этом примере). Точный размер можно указывать в пикселах или em-единицах.

Следующие два правила форматируют классы .Teaser и .Byline:

```
.Header .Teaser {
  margin:      0px;
  font-weight: bold;
}
```

```
.Header .Byline {
  font-style: italic;
  font-size:  small;
  margin:     0px;
}
```

Эти правила применяются для двух `<p>`-элементов колонтитула. Один элемент `<p>` содержит подзаголовок и принадлежит подклассу `Teaser` класса `Header`. А второй элемент `<p>` содержит информацию об авторе статьи и принадлежит подклассу `Byline` класса `Header`:

```
<div class="Header">
  <h1>How the World Could End</h1>
  <p class="Teaser">Scenarios that spell the end of life as we know</p>
  <p class="Byline">by Ray N. Carnation</p>
</div>
```

Следующее правило форматирует элемент `<div>`, принадлежащий классу `Content` и содержащий основное тело страницы. Правило устанавливает шрифт, поля между содержимым и границами окна и высоту строк:

```
.Content {
  font-size: medium;
  font-family: Cambria, Cochin, Georgia, "Times New Roman", Times, serif;
  padding-top:    20px;
  padding-right:  50px;
  padding-bottom: 5px;
  padding-left:   50px;
  line-height:    120%;
}
```

В то время как для колонтитула установлены одинаковые поля по всем сторонам, в содержимом для каждой стороны устанавливаются разные поля, добавляющие дополнительные интервалы вверху и еще большие по сторонам. Один из способов сделать это — использовать расширенные свойства поля (т. е. `padding-top`, `padding-right` и т. д.), как представлено в данном правиле. Другой способ — присвоить свойству `padding` список значений ширины каждого поля, но здесь нужно помнить правильный порядок этих полей — верхнее, правое, нижнее, левое:

```
padding:  20px 50px 5px 50px;
```

Обычно этот метод применяется, когда нужно указать поля для всех сторон, а если требуется задать поле только для некоторых сторон, применяется метод расширенных свойств `padding`. Но по большому счету, это всего лишь вопрос личных предпочтений программиста.

Последнее свойство — `line-height` — устанавливает интервал между смежными строками текста. Значение `120%` добавляет дополнительный промежуток между строк, что делает текст более удобным для чтения.

За правилом форматирования содержимого идут три контекстных селектора для внутреннего форматирования. Первое правило форматирует блок класса `LeadIn`, в частности, устанавливая полужирную капитель большого размера:

```
.Content .LeadIn {
  font-weight:  bold;
  font-size:    large;
  font-variant: small-caps;
}
```

Следующие два правила форматируют элементы `<h2>` и `<p>` в основном содержимом:

```
.Content h2 {
  color:         #24486C;
  margin-bottom: 2px;
  font-size:     medium;
}
.Content p {
  margin-top: 0px;
}
```

Как видно, хотя таблица стилей становится длиннее, она не обязательно усложняется. В ней просто повторяются те же основные методы (селекторы класса и контекстные селекторы) для форматирования разных частей документа.

Таблица стилей завершается тремя правилами для форматирования нижнего колонтитула. К этому времени вы должны уже понимать определяемое в нем форматирование:

```
.Footer {
  text-align: center;
  font-size:  x-small;
}
.Footer .Disclaimer {
  font-style: italic;
}
.Footer p {
  margin: 3px;
}
```

Ну вот, мы и рассмотрели таблицу стилей ApocalypsePage_Original.css. Можете загрузить ее с сайта книги (**http://www.prosetech.com/html5/**) и попробовать поэкспериментировать с ней, наблюдая получаемые изменения. Или же можете ознакомиться с материалом в *главе 2*, в котором эта страница и ее сопутствующая таблица стилей модифицируются для использования семантических элементов HTML5.

ПРИЛОЖЕНИЕ 2

2

Очень краткое введение в JavaScript

Было время, когда весь Интернет состоял исключительно из разметки. Страницы содержали только HTML-теги и текст, и ничего больше. По настоящему продвинутые веб-сайты использовали серверные сценарии, которые могли немного настроить разметку HTML перед тем, как отправлять ее браузеру. Но это всё.

Откройте страницу в текстовом редакторе сегодня, и вы, скорее всего, увидите кучи кода JavaScript, приводящего в движение все — от необходимых функциональностей до мелких примочек. Самозаполняющиеся поля ввода, всплывающие меню, слайд-шоу и электронная почта с веб-интерфейсом — вот лишь несколько примеров использования JavaScript хитроумными разработчиками. В действительности, современный Интернет практически невозможно представить без JavaScript. В то время как HTML остается каркасом Интернета, интеллектом, оживляющим большинство его продвинутых страниц, является JavaScript.

Это приложение предлагает ускоренный вводный курс в JavaScript. Оно не является полным руководством по использованию JavaScript и не содержит достаточно информации, чтобы позволить вам начать работать с этим языком, если у вас нет абсолютно никаких знаний другого языка программирования. Но если вы обладаете хоть малейшими знаниями программирования — например, когда-то немного программировали на Visual Basic, освоили основы Pascal или экспериментировали с языком C, тогда это приложение позволит вам перенести приобретенные навыки в мир JavaScript. Оно содержит как раз достаточно информации, чтобы позволить вам узнать знакомые концепты программирования, такие как переменные, циклы и условные переходы. Также в нем охватываются все основные элементы языка, которые используются в примерах на основе JavaScript в остальном материале этой книги.

Совет

Если вам требуется более полная помощь по JavaScript, можете найдите книгу о популярном инструментарии jQuery для расширения возможностей JavaScript. Или же прочитайте подробное, но сухое руководство по JavaScript от Mozilla, доступное по адресу **http://developer.mozilla.org/en/JavaScript/Guide**.

Принципы работы JavaScript в веб-странице

Чтобы код JavaScript правильно работал в веб-странице, он должен быть помещен в нее должным образом. В этом вопросе ключевым является элемент <script>. В последующих разделах мы рассмотрим разные способы оснащения страницы кодом JavaScript, от быстрой вставки небольших фрагментов кода непосредственно в любое место веб-страницы до тщательно структурированных программ, помещенных в отдельный файл.

Вставка кода в разметку

Самый простой способ добавления JavaScript-кода в страницу — это вставка его куда-либо в разметку HTML между тегами <script>:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>A Simple JavaScript Example</title>
</head>

<body>
  <p>At some point in the processing of this page, a script block
     will run and show a message box.</p>

  <script>
   alert("We interrupt this web page with a special " +
         "JavaScript announcement.");
  </script>

  <p>If you get here, you've already seen it.</p>
</body>
</html>
```

Данный сценарий содержит только одну строку кода, хотя он с таким же успехом мог бы иметь целый ряд операций. В данном случае эта строка кода активирует встроенную функцию JavaScript `alert()`. Эта функция принимает в качестве параметра строку текста, который и отображает в окне сообщения (рис. П2.1). Чтобы закрыть окно, нужно нажать кнопку **OK**.

ПРИМЕЧАНИЕ

В данном примере вводится соглашение JavaScript, которое применяется во всем материале этой книги и на профессиональных веб-сайтах: *конечная точка с запятой*. В языке JavaScript точка с запятой указывает на завершение оператора. Строго говоря, использование точки с запятой не является обязательным (если только вы не хотите влепить несколько операторов в одну строку), но считается хорошим стилем программирования.

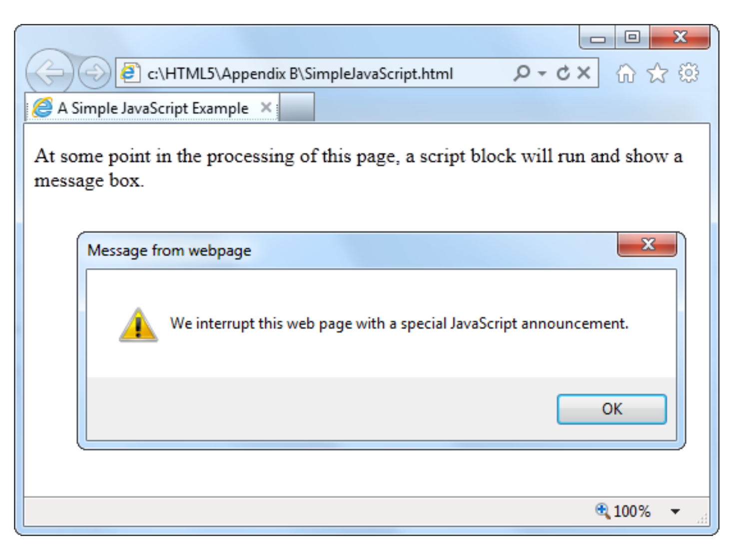


Рис. П2.1. Когда в процессе обработки веб-страницы браузер наталкивается на код JavaScript, он его немедленно исполняет. Более того, на время исполнения кода JavaScript временно прекращается вся другая обработка страницы. В данном примере дальнейшее исполнение кода приостановлено до тех пор, пока не будет должным образом закрыто окно сообщения, для чего нужно нажать кнопку **OK**. После этого код продолжит исполнение (в данном примере просто завершит исполнение), а браузер вернется к обработке следующей за кодом разметки

Как правило, разумно помещать код, исполняющийся с загрузкой страницы (как в нашем примере), в конце блока `<body>` непосредственно перед закрывающим тегом `</body>`. Таким образом, браузер будет исполнять его только после обработки всей разметки страницы.

МАЛОИЗВЕСТНАЯ ИЛИ НЕДООЦЕНЕННАЯ ВОЗМОЖНОСТЬ

Успокоительное для параноидного Internet Explorer

Рассмотренный пример исполняется без проблем в таких браузерах, как Firefox или Chrome, чего нельзя сказать об Internet Explorer. При попытке запустить его в этом браузере Internet Explorer выводит вверху окна желтую полосу с предупреждением о блокировании активного содержимого. До тех пор пока пользователь не щелкнул по этой полосе и в последующих диалогах не разрешил исполнение активного содержимого данным файлом, содержащийся в нем код JavaScript не будет исполняться.

С первого взгляда может показаться, что такая реакция Internet Explorer на безобидный фрагмент кода наверняка отпугнет от страницы потенциальных посетителей. Но не стоит беспокоиться по этому поводу, т. к. Internet Explorer проявляет эту бдительность только тогда, когда файл запускается локально. Когда та же страница предоставляется веб-сервером, то Internet Explorer не имеет против нее ни малейших возражений.

Но при всем этом вся процедура с предупреждениями и разрешениями исполнения является довольно раздражающей при тестировании веб-страницы. Эта проблема решается путем указания Internet Explorer считать, что страница предоставляется веб-сервером. Для этого в блок `<head>` страницы добавляется метка MOTW *(см. также разд. "Добавление JavaScript-кода" главы 1)*:

```
<head>
  <meta charset="utf-8">
```

```
  <!-- saved from url=(0014)about:internet -->
  ...
</head>
```

Браузер Internet Explorer обрабатывает страницы, содержащие метку MOTW, как будто бы они были предоставлены веб-сервером, исполняя содержащийся в них код JavaScript без проволочек. Для всех остальных браузеров эта метка выглядит просто как обычный комментарий HTML.

Использование функций

Проблема с предыдущим примером состоит в том, что такой подход смешивает разметку и код в одну кучу. Чтобы навести в странице некоторый порядок, код для выполнения отдельной задачи следует помещать в *функцию* — именованный фрагмент кода, который исполняется по требованию.

Будет хорошей идеей присваивать функции логическое название, отображающее ее назначение. Вот пример создания функции:

```
function showMessage() {
  // Код вставляется сюда ...
}
```

На данном этапе эта функция содержит лишь один комментарий и никакого кода. (В JavaScript однострочные комментарии обозначаются двумя косыми чертами. Браузеры игнорируют строки, начинающиеся этими символами.)

Код для выполнения требуемой задачи вставляется между фигурными скобками, как показано в следующем примере:

```
function showMessage() {
  alert("We interrupt this web page with a special " +
        "JavaScript announcement.");
}
```

Конечно же, все это должно заключаться в блок `<script>`. Лучше всего размещать функции JavaScript в блоке `<head>` страницы. Таким образом, мы придаем странице организованность, убрав код из разметки и поместив его в отдельный блок страницы.

Далее приведен предыдущий пример, модифицированный для использования функции:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>A Simple JavaScript Example</title>
  <script>
    function showMessage() {
      alert("We interrupt this web page with a special " +
            "JavaScript announcement.");
    }
```

```
  </script>
</head>
...
```

Сами по себе функции ничего не делают. Чтобы исполнить код в функции, ее нужно *вызвать* другим фрагментом кода.

Вызов функции не представляет ничего сложного; мы уже видели, как это делается в случае со встроенной функцией `alert()`. Чтобы вызвать функцию, мы просто пишем ее имя, а потом пару круглых скобок. В скобках функции передаются данные, которые могут потребоваться ей. Если же функции не нужно передавать никаких данных, как для нашей функции `showMessage()`, то скобки просто оставляются пустыми:

```
<body>
  <p>At some point in the processing of this page, a script block
     will run and show a message box.</p>
  <script>
    showMessage();
  </script>
  <p>If you get here, you've already seen it.</p>
</body>
</html>
```

С первого взгляда может показаться, что поскольку этот новый подход использует два блока кода, он усложняет код. Но в действительности это большой шаг вперед по нескольким причинам, включая следующие.

- **Основная масса кода теперь удалена из разметки.** Все, что осталось от кода в разметке теперь, — это только одна строка для вызова функции. В отличие от функции в нашем примере, настоящая функция будет содержать кучу кода, а настоящая страница — много функций. Поэтому удаление всех этих подробностей из разметки, несомненно, улучшит ее.
- **Код можно повторно использовать.** Поместив код для выполнения определенной задачи в функцию, мы можем вызывать этот код неограниченное количество раз из разных мест страницы. В данном примере это не так очевидно, но эта возможность является важным фактором в более сложных приложениях, наподобие приложения для рисования, рассматриваемого в *главе 6*.
- **Код можно поместить во внешние файлы.** Удаление кода из разметки в отдельный блок страницы — это шаг по направлению к удалению его из страницы вообще и помещению в отдельный файл. Это действие по улучшению организации страницы рассматривается в следующем разделе.
- **Возможность добавлять события.** Событие — это метод, посредством которого странице можно дать указание выполнить определенную функцию, когда происходит определенное действие. Веб-страницы движимы событиями. Это означает, что большинство кода исполняется, когда происходит какое-либо событие (а не запускается из блока кода JavaScript). Для событий требуются функ-

ции, и события позволяют активировать функции, не прибегая к дополнительному блоку кода, как мы увидим в *разд. "Реагирование на события" далее в этом приложении*.

ПРИМЕЧАНИЕ

В страницу можно вставить любое количество блоков `<script>`.

Перемещение кода JavaScript в файл сценариев

Собрав весь код JavaScript в набор функций и поместив их в отдельном блоке страницы, мы сделали первый шаг по направлению к упорядочиванию содержимого веб-страницы. Вторым шагом будет удаление всего этого кода из веб-страницы вообще и помещение его в отдельном файле. Это позволит нам получить более простые веб-страницы меньшего объема, а также возможность использовать одни и те же функции в разных веб-страницах. По сути, помещение кода сценариев в отдельный файл аналогично помещению в отдельный файл правил стилей CSS. В обоих случаях мы получаем возможность повторного использования нашего кода и создаем более простые страницы.

ПРИМЕЧАНИЕ

Практически все веб-страницы с хорошим дизайном, в которых используются сценарии JavaScript, помещают этот код в один или несколько отдельных файлов. Единственным исключением будет несколько строк предельно простого кода, который достоверно не будет использоваться нигде больше или же в случае единичного образца сценария.

Файл сценария — это простой текстовый файл с расширением js (обозначающим JavaScript). Код помещается в файл сценариев таким же образом, как и в файл веб-страницы, но без использования элемента `<script>`. В качестве примера далее приведено полное содержимое файла сценариев MessageScripts.js:

```
function showMessage()  {
  alert("We interrupt this web page with a special " +
        "JavaScript announcement.");
}
```

Сохраните этот файл в той же папке, что и веб-страницу. В блоке `<head>` веб-страницы определяем блок `<script>`, но не помещаем в него никакого кода. Вместо этого вставляем в блок атрибут `src` и в качестве значения присваиваем ему имя нашего файла сценариев:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>A Simple JavaScript Example</title>
  <script src="MessageScripts.js"></script>
</head>
```

```
<body>
  <p>At some point in the processing of this page, a script block
     will run and show a message box.</p>
  <script>
    showMessage()
  </script>
  <p>If you get here, you've already seen it.</p> </body> </html>
```

Когда браузер обнаружит этот блок сценариев, он запросит файл MessageScripts.js и будет рассматривать содержащийся в нем код, как будто бы тот находится в веб-странице. Это означает, что функцию `showMessage()` можно вызывать точно таким же образом, как мы это делали ранее.

Примечание

Хотя при использовании отдельных файлов сценариев блок `<script>` в действительности не содержит никакого кода, его все равно нужно закрывать тегом `</script>`. В противном случае браузер будет полагать, что все, что следует за открывающим тегом `<script>`, т. е. остальная часть страницы, является частью JavaScript-кода.

Страницу можно связать с файлом сценариев на другом веб-сайте. Для этого вместо простого имени файла атрибуту `src` нужно присвоить полный URL файла (например, **http://SuperScriptSite.com/MessageScript.js**). Эта технология позволяет подключать услуги других веб-компаний, например Google Maps *(см. разд. "Отображение карты" главы 12)*.

Реагирование на события

На данном этапе мы рассмотрели только один способ исполнения сценария — в ходе загрузки страницы. Но намного чаще требуется исполнять код после завершения загрузки страницы, когда пользователь выполняет какое-либо действие, например нажимает кнопку или наводит указатель мыши на какой-либо элемент страницы.

Для этого нужно использовать *события* (events) JavaScript, которые представляют извещения, отправляемые элементом HTML в ответ на определенные развития. Так, каждый элемент HTML имеет JavaScript-событие `onMouseOver`, которое срабатывает при наведении указателя мыши на элемент, например блок текста, ссылку, изображение, ячейку таблицы, текстовое поле и т. п. В результате срабатывания события исполняется связанный с ним код.

Здесь возникает естественный вопрос: а каким образом код связывается с событием элемента? Для этого необходимое событие нужно добавить в элемент в качестве атрибута, а в качестве значения этого атрибута указать требуемую функцию. Таким образом, если мы хотим связать определенный код с событием `onMouseOut` элемента `<img>`, то можем это сделать так:

```
<img src="sunny.jpg" alt="A sunny day" onmouseover="showMessage()">
```

ПРИМЕЧАНИЕ

В JavaScript имена функций, переменных и объектов являются чувствительными к регистру, что означает, что `showMESSAGE` — не одно и то же, что `showMessage`. Но имена атрибутов событий не чувствительны к регистру, т. к. они технически являются частью разметки HTML, который относится терпимо к любой комбинации букв верхнего и нижнего регистра. Тем не менее, распространенной практикой является написание атрибутов событий без прописных букв (как показано в примере ранее), т. к. это соответствует старым правилам XHTML; в любом случае, большинство программистов слишком ленивые, чтобы постоянно тянуться к клавише <Shift>.

Теперь наведение указателя мыши на изображение активирует событие `onMouseOver`, в ответ на которое браузер вызывает функцию `showMessage()`, выводящую уже знакомое окно сообщения (рис. П2.2). Функция, вызываемая в ответ на срабатывание события, называется *обработчиком события*.

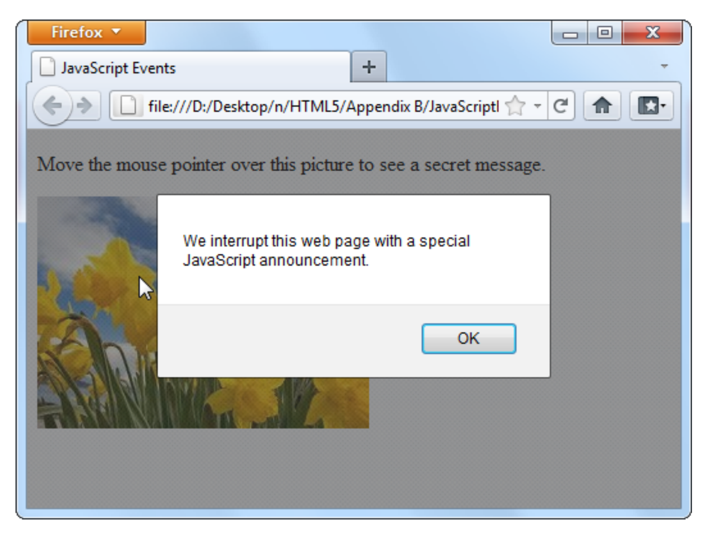


Рис. П2.2. В этом примере окно сообщения выводится не при загрузке страницы, а в любое время после того, как пользователь наведет указатель мыши на изображение

Чтобы эффективно использовать события, нужно знать, какие события поддерживаются в JavaScript. Кроме того, нужно знать, какие события с какими элементами HTML работают. В табл. П2.1 перечислены наиболее часто используемые события и элементы HTML, к которым они применимы. Более полное справочное руководство см. по адресу **http://developer.mozilla.org/en/DOM/element**.

Таблица П2.1. *Наиболее употребляемые события объектов HTML*

Название события	Срабатывает	Применимо
`onClick`	При щелчке по элементу	Практически ко всем элементам
`onMouseOver`	При наведении указателя мыши на элемент	Практически ко всем элементам

Таблица П2.1 *(окончание)*

Название события	Срабатывает	Применимо
onMouseOut	Когда указатель мыши убирается с элемента	Практически ко всем элементам
onKeyDown	При нажатии клавиши	<select>, <input>, <textarea>, <a>, <button>
onKeyUp	Когда отпускается нажатая клавиша	<select>, <input>, <textarea>, <a>, <button>
onFocus	При получении фокуса элементом (т. е. при выделении элемента мышью, клавишей <Tab> или клавишами стрелок). Элементы включают текстовые поля, флажки и т. п. (см. табл. 4.1 для дополнительной информации)	<select>, <input>, <textarea>, <a>, <button>
onBlur	Когда элемент теряет фокус	<select>, <input>, <textarea>, <a>, <button>
onChange	При изменении значения элемента ввода. В случае текстового поля срабатывает только после перехода на другой элемент	<select>, <input type="text">, <textarea>
onSelect	При выборе части текста в элементе ввода	<inputtype="text">, <textarea>
onError	При ошибке загрузки браузером изображения (обычно по причине неправильного URL)	<img>
onLoad	По окончанию загрузки браузером новой страницы или объекта, такого, как изображение	<img>, <body>
onUnload	Когда браузер выгружает страницу. Это обычно происходит при переходе на новый URL в строке адреса или по ссылке. Срабатывание происходит *до того*, как браузер загрузит новую страницу	<body>

Несколько основных структур языка JavaScript

Краткого приложения, наподобие этого, недостаточно, чтобы охватить все аспекты любого языка программирования, даже такого простого, как JavaScript. Но в последующих разделах мы рассмотрим несколько основных конструкций языка, необходимых для понимания примеров, приводимых в этой книге.

Переменные

Все языки программирования имеют понятие *переменных* — именованных контейнеров, используемых для хранения информации в памяти. В JavaScript переменные

создаются с помощью ключевого слова `var`, за которым следует имя переменной. Это единственный способ создания переменных в JavaScript. Вот пример создания переменной `myMessage`:

```
var myMessage;
```

> **ПРИМЕЧАНИЕ**
>
> Имена переменных в JavaScript чувствительные к регистру, что означает, что переменная `myMessage` — это совсем другая переменная, чем переменная `MyMessage`. При попытке использовать первую вместо второй или наоборот в лучшем случае браузер выдаст сообщение об ошибке, а в худшем, более частом, случае в странице произойдет какая-либо странная ошибка.

Данные сохраняются в переменной посредством знака равенства (=), который копирует данные из правой его части в переменную, указанную в левой части равенства. Далее приведен пример объявления переменной и присвоения ей текстового значения (которое в программировании называется *строкой*):

```
var myMessage = "Everybody loves variables";
```

Эту переменную потом можно использовать в коде:

```
// Отображаем значение переменной в окне сообщения.
alert(myMessage);
```

> **ПРИМЕЧАНИЕ**
>
> Язык JavaScript печально известен своей вседозволенностью и разрешает использовать переменные, даже если они не были объявлены заранее с помощью ключевого слова `var`. Но это чрезвычайно плохая практика, которая может вылиться в нелепые ошибки.

Значение *null*

В JavaScript переменным может присваиваться специальное значение `null`, которое означает отсутствие значения. Значение переменной, равное `null`, означает, что данного объекта не существует. В зависимости от контекста это может указывать на то, что определенная возможность недоступна. Например, Modernizr *(см. разд. "Определение возможностей с помощью Modernizr" главы 1)* использует тестирование на `null`, чтобы определить поддержку браузером определенных возможностей HTML5. Тестирование на `null` можно применять и в своих сценариях, например, чтобы определить, был ли уже создан или сохранен объект:

```
if (myObject == null) {
  // Объект myObject не существует.
  // Теперь, может быть, удачное время, чтобы создать его.
}
```

Область видимости переменных

Переменные можно создавать в двух основных местах — внутри функции и вне функции. Следующий фрагмент кода содержит оба эти типа переменных:

```
<script>
  var outsideVariable;

  function doSomething() {
    var insideVariable;
    ...
  }
</script>
```

Созданная внутри функции переменная называется *локальной переменной* и существует только на протяжении времени исполнения этой функции. В приведенном примере переменная `insideVariable` является локальной. Как только метод `doSomething()` завершает исполнение, эта переменная удаляется из памяти. Это означает, что при следующем вызове метода `doSomething()` переменная `insideVariable` создается заново и получает новое значение.

С другой стороны, значение переменной, созданной вне функции (такие переменные называются *глобальными*), сохраняется на протяжении всего времени, пока страница загружена в браузере. Вдобавок такие переменные могут использоваться всеми функциями. В приведенном примере переменная `ousideVariable` является глобальной.

СОВЕТ

Используйте локальные переменные за исключением случаев, когда переменную нужно сделать доступной для нескольких функций или же сохранить ее значение после исполнения функции. Отслеживание глобальных переменных требует больше усилий и большое количество таких переменных загромождает код.

Типы данных переменных

Переменные JavaScript могут содержать разные типы данных, такие как текст, целые числа, числа с плавающей запятой и объекты. Но независимо от типа данных, которые вы намереваетесь хранить в переменной, все они создаются посредством одного и того же ключевого слова `var`. При этом тип данных для конкретной переменной *не* указывается.

Это означает, что можно взять переменную `myMessage`, которая содержит текстовую строку, и присвоить ей числовое значение:

```
myMessage = 27.3;
```

Такой подход облегчает использование языка JavaScript, т. к. любая переменная может хранить данные любого типа. Но в то же самое время это способствует возникновению ошибок. Например, мы хотим взять текстовую строку из поля ввода и сохранить ее в переменной:

```
var = inputElement.value;
```

Но если не соблюдать должной осторожности, можно случайно сохранить в этой переменной не содержимое объекта, а сам объект:

```
var = inputElement;
```

Так как JavaScript позволяет выполнять оба типа присваиваний и не может угадывать мыслей программиста, он воспримет последнее присвоение как должное. Но дальше в коде эта оплошность, скорее всего, вызовет какую-либо неисправимую ошибку исполнения. Браузер просто прекратит выполнение оставшегося кода и не выдаст вам никаких сообщений об ошибке, объясняющих ее причину.

Арифметические операции

Простая возможность присваивать переменным значения вряд ли объясняла бы надобность в них. По настоящему полезными переменные делает возможность выполнять над ними *операции*, модифицирующие содержащиеся в них данные. Например, с переменными можно применять арифметические операторы для выполнения математических вычислений:

```
var myNumber = (10 + 5) * 2 / 5;
```

Вычисления выполняются в стандартном порядке — сначала в скобках, затем умножение и деление, а потом сложение и вычитание. Результатом этих операций будет `6`.

С помощью операторов можно также конкатенировать несколько текстовых строк в одну длинную строку. Для этого используется оператор "плюс" (+):

```
var firstName = "Sarah";
var lastName = "Smithers";
var fullName = firstName + " " + lastName;
```

В результате исполнения приведенного кода переменная `fullName` будет содержать текстовое значение `"Sarah Smithers"`. (Пробел в кавычках (`" "`) используется для того, чтобы оставить промежуток между именем и фамилией.)

Простые арифметические операции можно обозначать с помощью сокращенной нотации. Например, следующую простую операцию сложения

```
var myNumber = 20;
myNumber = myNumber + 10;
// (Теперь значение переменной myNumber равно 30.)
```

можно записать так:

```
var myNumber = 20;
myNumber += 10;
// (Теперь значение переменной myNumber равно 30.)
```

Прием с переносом оператора на левую сторону знака равенства работает, по большому счету, со всеми операторами. Вот еще несколько примеров:

```
var myNumber = 20;
myNumber -= 10;
// (Теперь значение переменной myNumber равно 10.)
myNumber *= 10;
// (Теперь значение переменной myNumber равно 100.)
```

```
var myText = "Hello";
var myText += " there.";
// (Теперь значение переменной myText равно "Hello there.")
```

А добавить или вычесть единицу можно еще более простым способом:

```
var myNumber = 20;
myNumber++;
// (Теперь значение переменной myNumber равно 21.)

myNumber--;
// (Теперь значение переменной myNumber равно 20.)
```

АВАРИЙНАЯ СИТУАЦИЯ
Поиск ошибок в коде JavaScript

Чтобы находить и исправлять ошибки в коде JavaScript (наподобие ошибки присвоения переменной объекта вместо строки, рассмотренной ранее), нужно научиться выполнять отладку — выявление причин проблем в коде и их устранение. К сожалению, способ выполнения отладки зависит от используемого браузера, и разные браузеры имеют различные отладочные средства (или поддерживают разные отладочные модули расширения). И хотя все эти средства предназначены для выполнения одинаковой задачи, они применяют для этого не совсем одинаковые методы.

К счастью, в сети можно найти всю требуемую информацию для выполнения отладки на разных браузерах. Далее приводятся ссылки на некоторые ресурсы.

- **Internet Explorer.** Чтобы начать процесс отладки, нажмите клавишу <F12>; откроется окно **Средства разработчика** (Developer Tools). Подробные инструкции по использованию этого средства см. здесь: **http://msdn.microsoft.com/ie/aa740478**.
- **Firefox.** Серьезные разработчики платформы Firefox используют модуль расширения Firebug (**http://getfirebug.com/javascript**), который позволяет наблюдать за исполнением кода в любое время. Кроме этого, на сайте **http://developer.mozilla.org/en/Debugging_JavaScript** можно ознакомиться с документацией от Mozilla по средствам отладки.
- **Google Chrome.** Браузер Chrome имеет приличный встроенный отладчик. Обучающее пособие по его применению доступно по адресу **http://code.google.com/chrome/extensions/tut_debugging.html**.
- **Opera.** Браузер Opera имеет встроенный отладчик Dragonfly (см. описание по адресу **www.opera.com/dragonfly**), а по адресу **http://tinyurl.com/39nv7w** можно найти хороший обзор основных методов отладки.
- **Safari.** Браузер Safari имеет мощный набор встроенных средств отладки, хотя найти документацию по ним, может быть, несколько проблематично. Для начала ознакомьтесь с технической статьей из библиотеки Safari Developer Library по адресу **http://tinyurl.com/63om77c**.

Помните, что не имеет значения, каким браузером и отладочными средствами пользоваться, чтобы исправить проблемы со страницей. Когда они исправлены, то исправлены для всех браузеров.

Условные переходы

Все условные переходы начинаются с *условия*: выражения, значение которого может быть или `true` (истина), или `false` (ложь). В зависимости от значения выражения принимается решение, выполнять определенную часть кода или пропустить ее.

Условия создаются с помощью операторов сравнения и логических операторов (табл. П2.2).

Таблица П2.2. *Логические операторы*

Оператор	**Описание**
`==`	Равно
`!=`	Не равно
`!`	`Not` (не, этот унарный оператор изменяет значение своего операнда на противоположное, т. е. с истины на ложь и наоборот)
`<`	Меньше
`>`	Больше
`<=`	Меньше или равно
`>=`	Больше или равно
`&&`	Логическое "И". Вычисляется, только если значения обоих операндов равны `true` (истина). Если значение первого операнда равно `false` (ложь), значение второго операнда не вычисляется
`||`	Логическое "Или". Вычисляется, если значение любого из операндов равно `true`. Если значение первого операнда равно `true`, значение второго операнда не вычисляется

Далее приведен пример простого сравнения:

```
myNumber < 100
```

Чтобы это сравнение превратить в условие, его надо использовать в операторе условного перехода `if`. Вот пример такого оператора:

```
if (myNumber < 100)  {
  // (Этот код исполняется, если значение переменной myNumber
  // равно 20, и не исполняется, если оно равно 147.)
}
```

Примечание

Вообще, код, исполняющийся, когда условие истинно, не обязательно заключать в фигурные скобки, если только он не состоит из нескольких операторов. Но использование этих скобок всегда делает код более понятным и помогает избежать возможных ошибок в случае нескольких операторов.

При проверке на равенство всегда используются *два знака равенства*, т. е. ==. *Один знак равенства* является оператором присваивания и устанавливает значение переменной, а не выполняет требуемое нам сравнение:

```
// Правильно:
if (myName == "Doe")  {
}
```

```
// Неправильно:
if (myName = "Sarah")  {
}
```

Чтобы оценить несколько условий, одно за другим, используется несколько блоков `if`. Но если среди нескольких условий нужно выбрать одно, игнорируя прочие, к оператору `if` добавляется ключевое слово `else`. Далее приведен пример использования оператора `else`:

```
if (myNumber < 100)  {
  // (Этот код исполняется, если значение переменной myNumber меньше 100.)
}
else if (myNumber < 200) {
  // (Этот код исполняется, если значение переменной myNumber меньше 200,
  // но больше или равно 100.)
}
else {
  // (Этот код исполняется во всех других случаях, т. е. когда
  // значение переменной myNumber равно или больше 200.)
}
```

Блок `if` может содержать любое количество условий, а использование конечного `else` без условия не является обязательным.

Циклы

Цикл — это основной инструмент программирования, позволяющий повторно исполнять блок кода. Основным циклом в языке JavaScript является цикл `for`, который имеет встроенный счетчик. Большинство языков программирования имеет свои версии этой управляющей структуры.

При создании цикла `for` устанавливается начальное значение счетчика, конечное значение и шаг инкремента после каждого прохода. Вот пример цикла `for`:

```
for (var i = 0; i < 5; i++){
  // (Этот код исполняется пять раз.)
  alert("This is message: " + i);
}
```

В начале цикла в круглых скобках указываются данные счетчика. Сначала создается переменная счетчика и ей присваивается значение `0` — `var i = 0`. Потом указывается конечное значение счетчика, т. е. условие для завершения цикла — `i < 5`. Если условие не удовлетворяется, например значение `i` равно `5`, цикл завершается, и код внутри больше не исполняется. Последнее выражение — `i++` — увеличивает значение переменной счетчика на единицу при каждом проходе цикла. (Все эти выражения разделяются точкой с запятой.) Это означает, что для первого прохода цикла значение `i` будет `0`, для второго `1` и т. д. В итоге код цикла исполняется пять раз и выводит такую последовательность сообщений:

```
This is message: 0
This is message: 1
```

```
This is message: 2
This is message: 3
This is message: 4
```

Массивы

Цикл естественно сочетается с *массивом* — объектом программирования, в котором сохраняется несколько значений.

Массивы JavaScript обладают высшей степенью гибкости. В отличие от других языков программирования, при объявлении массива в JavaScript его размер не указывается. Массив создается с помощью ключевого слова `var`, за которым следует имя массива:

```
var colorList = [];
```

Значения последовательным элементам массива присваиваются посредством метода `add()` объекта массива:

```
colorList.add("blue");
colorList.add("green");
colorList.add("red");
```

Можно также присвоить значение определенному элементу массива. Если данный элемент не существует, он создается автоматически:

```
colorList[3] = "magenta";
```

Можно извлекать значения из конкретных элементов массива, присваивая их обычным переменным:

```
var color = colorList[3];
```

> **Примечание**
>
> Следует иметь в виду, что в JavaScript счет ведется с *нуля*: номер первого элемента массива равен 0, второго — 1 и т. д.

Элементы массива можно обрабатывать, используя цикл `for`:

```
for (var i = 0; i < colorList.length; i++) {
  alert("Found color: " + colorList[i]);
}
```

Этот код обрабатывает элементы массива от первого (ячейка с номером `0`) до последнего (номер которого определяется посредством свойства массива `length`, которое возвращает общее количество элементов массива). Код отображает значение каждого элемента массива в окне сообщений, хотя, несомненно, можно было бы придумать более практичное применение.

Использование цикла `for` для обработки массивов является одним из основных методов в JavaScript. В книге мы будем часто использовать этот метод, как с массивами, создаваемыми нами, так и с массивами, предоставляемыми другими функциями JavaScript.

Функции, которые получают и возвращают данные

Ранее мы рассмотрели простую функцию `showMessage()`. При вызове этой функции ей не нужно предоставлять никаких данных, а по завершению исполнения она не возвращает никакой информации.

Функции не всегда такие простые. Во многих случаях функциям нужно передавать информацию или же получить результаты исполнения функции и использовать их для другой операции. Допустим, что нам нужно создать версию функции `showMessage()`, которая может показывать разные сообщения. Для этого функцию `showMessage()` нужно модифицировать так, чтобы она принимала *параметр*. Этот параметр представляет любой текст, который мы хотим отображать посредством этой функции.

Чтобы оснастить функцию параметром, ему нужно присвоить имя, например `customMessage`, и поместить его в круглые скобки после названия функции:

```
function showMessage(customMessage) {
  alert(customMessage);
}
```

Примечание

Функция может принимать любое количество параметров. Множественные параметры в списке разделяются *запятыми*.

Внутри функции параметры используются, как обычные переменные. В этом примере функция просто принимает предоставленный ей текст и отображает его в окне сообщения.

Теперь при вызове функции `showMessage()` ей нужно передать значение (которое называется *аргументом*[1]) для каждого параметра функции:

```
showMessage("Nobody likes an argument.");
```

Таким образом, параметры позволяют передавать информацию функции. Можно также создать функцию, отправляющую информацию *назад* коду, который вызвал ее. Для этого с функцией используется команда `return`, которая помещается в самом конце функции. Эта команда немедленно прекращает выполнение функции и возвращает созданную функцией информацию.

Конечно же, продвинутая функция может принимать *и* возвращать информацию. Так, далее приводится пример функции, которая умножает два передаваемых ей числа (параметры `numberA` и `numberB`) и возвращает полученный результат:

```
function multiplyNumbers(numberA, numberB) {
  return numberA * numberB;
}
```

[1] Часто термины *"параметр"* и *"аргумент"* используют взаимозаменяемо, но между ними есть разница. Вкратце эту разницу можно определить следующим образом — параметры принимаются, а аргументы передаются.

Вот пример, как эту функцию можно использовать на странице:

```
// Передаем функции два числа и получаем результат.
var result = multiplyNumbers(3202, 23405);

// Используем результат, чтобы создать сообщение.
var message =  "The product of 3202 and 23405 is  " + result;

// Отображаем сообщение.
showMessage(message);
```

Конечно же, для того чтобы умножить два числа, нам функция не нужна (это можно сделать посредством обычных операций JavaScript), также как и не нужна нам функция для отображения окна сообщения (поскольку встроенная функция `alert()` может прекрасно справиться с этой задачей). Но эти примеры служат хорошей иллюстрацией работы и применения функций, и мы будем использовать параметры и возвращаемые значения точно таким же образом в более сложных функциях.

Взаимодействие со страницей

Итак, мы знаем, как правильно вставить код JavaScript в страницу, но кроме вывода окна сообщения мы еще не применили эти знания для выполнения какой-либо практической задачи. И прежде чем начать двигаться в этом направлении, нам нужно узнать немного больше о типичной роли JavaScript.

Прежде всего, важно понимать, что код JavaScript исполняется в "песочнице". Это означает, что его возможности тщательно ограничиваются. Этот код не может выполнять задачи, потенциально угрожающие безопасности компьютера посетителя, такие как, например, управление принтером, обращение к файлам, исполнение других программ, форматирование жесткого диска и т. п. Такой подход позволяет хорошо обезопасить компьютер даже самых неосторожных посетителей.

Большую часть времени JavaScript выполняет следующие задачи.

- **Обновляет страницу.** Код сценария может изменять, удалять и добавлять элементы страницы. Более того, JavaScript-код может изменить каждый аспект текущей отображаемой страницы и даже полностью заменить весь документ.
- **Получает данные с сервера.** Сценарий JavaScript может запрашивать данные с того же самого веб-сервера, который предоставил исходную страницу. Сочетая этот метод с предыдущим, можно создавать веб-страницы, которые плавно обновляют важную информацию, например список новостей или биржевые котировки.
- **Отправляет данные на веб-сервер.** В HTML уже имеется способ отправки данных с использованием веб-форм *(см. главу 4)*, но в JavaScript применяется более тонкий подход. Он позволяет снимать данные с элементов формы, проверять действительность этих данных и даже отправлять их на веб-сервер, и при этом не заставляя браузер обновлять страницу.

Для реализации последних двух видов задач требуется использовать объект `XMLHttpRequest`, расширение JavaScript, которое рассматривается в *разд. "Объект XMLHttpRequest" главы 11*. В последующих разделах мы рассмотрим первую из перечисленных задач, которая является главной частью почти каждой веб-страницы, движимой JavaScript.

Манипулирование элементами

С точки зрения JavaScript веб-страница является бо́льшим, чем просто статическим блоком кода HTML. Для этого кода каждый элемент разметки является объектом, который можно исследовать и модифицировать.

Самый простой способ получить доступ к объекту — это присвоить ему однозначное имя, что реализуется с помощью атрибута `id` объекта. Вот пример присвоения имени объекту:

```
<h1 id="pageTitle">Welcome to My Page</h1>
```

Элемент, которому присвоено однозначное имя, можно с легкостью найти в коде и манипулировать им посредством кода JavaScript.

Для поиска объектов в JavaScript применяется метод `getElementById()` объекта `document`. По большому счету, `document` — это объект, который представляет весь документ HTML. Он всегда доступен, и его можно использовать в любое время. Этот объект документа имеет значительное количество полезных свойств и методов. Метод `getElementById()` является одним из самых полезных — с его помощью можно просканировать веб-страницу, чтобы найти требуемый элемент HTML.

Примечание

Если вы знакомы с основами объектно-ориентированного программирования, то должны знать, что такое свойства и методы. Если же нет, то свойства можно представить как прикрепленные к объекту данные, а методы, как встроенные в объект функции.

При вызове метода `document.getElementById()` ему передается идентификатор элемента HTML, который требуется найти. В следующем примере показан код для получения элемента HTML с идентификатором `pageTitle`:

```
var titleObject = document.getElementById("pageTitle");
```

Этот код получает объект элемента `<h1>`, который был показан ранее, и сохраняет его в переменной `titleObject`. Над сохраненным в переменной объекте можно выполнять разные операции, не требуя поиска его каждый раз для каждой операции.

Какие же операции можно выполнять над объектами HTML? В некоторой степени, ответ зависит от типа элемента, с которым мы работаем. Например, если объект — гиперссылка, можно изменить его URL, а если объект — изображение, то можно изменить его источник. Некоторые операции можно выполнять почти со всеми элементами HTML, например, изменять стиль или текст между открывающим и закрывающим тегами. Как мы увидим далее, эти возможности позволяют сделать

страницы динамическими, например, изменить внешний вид страницы, когда посетитель исполняет на ней какое-либо действие, скажем, щелкает по ссылке. Подобные ответные действия вызывают у посетителя чувство взаимодействия с интеллектуальной, реагирующей программой, а не простой, инертной веб-страницей.

Далее приведен пример, как можно модифицировать текст нашего подопытного элемента `<h1>`:

```
titleObject.innerHTML = "This Page Is Dynamic";
```

Этот код присваивает свойству `innerHTML` объекта `titleObject` (это наш элемент `<h1>`) текстовое значение, которое отображается как содержимое этого элемента. Свойство `innerHTML` — это всего лишь один аспект объекта HTML, который можно изменять. Для создания кода для работы со свойствами объектов нужно знать, какие свойства JavaScript разрешает модифицировать.

Очевидно, что некоторые свойства относятся только к определенным элементам HTML, например атрибут `src`, который используется для загрузки изображения в элемент `<img>`:

```
var imgObject  = document.getElementById("dayImage");
dayImage.src =  "cloudy.jpg";
```

А посредством свойств объекта `style` можно изменять свойства CSS:

```
titleObject.style.color = "rgb(0,191,255)";
```

Современные браузеры поддерживают огромный список свойств DOM, которые можно использовать практически со всеми элементами HTML. Некоторые из наиболее полезных свойств приведены в табл. П2.3.

Таблица П2.3. *Наиболее употребляемые свойства объектов HTML*

Свойство	**Описание**
`className`	Позволяет получить или установить атрибут `class` *(см. разд. "Форматирование элементов посредством классов" приложения 1)*. Иными словами, это свойство определяет применяемые элементом стили (если таковые используются). Конечно же, этот стиль нужно сначала определить во встроенной или связанной со страницей таблице стилей, иначе получится простое форматирование по умолчанию
`innerHTML`	Позволяет считывать или изменять содержимое элемента HTML. Свойство `innerHTML` очень полезно, но имеет две особенности. Первая: его можно использовать на всем содержимом HTML, включая текст и теги. Поэтому вставить в абзац полужирный текст можно, присвоив свойству `innerHTML` значение `<b>Hi</b>`. Вторая: присвоенное свойству `innerHTML` новое значение затирает все старое содержимое этого элемента, включая все другие элементы HTML. Таким образом, если присвоить новое значение свойству `innerHTML` элемента `<div>`, который содержит несколько абзацев текста и изображений, все это содержимое будет заменено новым содержимым
`parentElement`	Предоставляет HTML-объект элемента, содержащего текущий элемент. Например, если текущий элемент является элементом `<b>` в абзаце `<p>`, это свойство возвращает объект этого элемента `<p>`. Полученный таким образом объект также можно модифицировать

Таблица П2.3 (окончание)

Свойство	Описание
style	Объединяет все атрибуты CSS, которые определяют внешний вид элемента HTML. Свойство style возвращает полноценный объект style и для изменения атрибута стиля нужно добавить еще одну точку, а потом название этого атрибута. Например, myElement.style.fontSize. С помощью объекта style можно устанавливать цвет, границы, шрифт и даже позиционирование элемента
tagName	Предоставляет название элемента HTML для данного объекта без угловых скобок. Например, значением tagName элемента <img> будет текст img

Совет

Элементы HTML также имеют небольшой набор полезных методов, включая методы для модифицирования атрибутов, такие как getAttribute() и setAttribute(), а также для добавления или удаления элементов, такие как insertChild(), appendChild() и removeChild(). Дополнительную информацию о свойствах и методах, поддерживаемых определенными элементами HTML, см. в справочной документации от Mozilla по адресу **http://developer.mozilla.org/en/DOM/element**.

Динамическое подключение к событию

В *разд. "Реагирование на события" ранее в этом приложении* мы рассмотрели, как запустить функцию на исполнения с помощью атрибута события. Но событие также можно подключить к функции, используя код JavaScript.

В большинстве случаев, скорее всего, для подключения события к коду следует использовать атрибуты событий. Но в некоторых случаях этот подход невозможен или неудобен. Одним из самых распространенных примеров будет создание объекта HTML в коде и последующее динамическое добавление его в страницу. В данной ситуации разметка для нового элемента отсутствует, вследствие чего использование атрибута события не возможно. (Этот метод применяется в примере рисования на холсте в *главе 6*.) Другим примером будет подключение события к встроенному объекту, а не к элементу. (Этот случай рассматривается в примерах обработки событий хранилища в *главе 9*.) По всем этим причинам важно понимать, как подключать события к коду.

Примечание

Существуют разные способы подключения событий, но не все они поддерживаются всеми браузерами. В этом разделе применяется подход свойства события, который поддерживается всеми браузерами. Кстати, если вы решите использовать какой-либо инструментарий JavaScript, наподобие jQuery, то, скорее всего, обнаружите, что он предоставляет еще одну систему подключений событий, которая *будет* работать на всех браузерах и может иметь несколько дополнительных возможностей.

К счастью, подключение событий не представляет ничего сложного. Нужно просто установить свойство события с таким же именем, как и атрибут события, который бы использовался в обычной ситуации.

Например, пусть где-то на странице есть элемент `<img>`:

```
<img id="dayImage" src="sunny.jpg" alt="The weather">
```

Браузеру можно дать указание вызвать метод `swapImage()` при щелчке мышью по этому изображению с помощью следующего кода:

```
var imgObject = document.getElementById("dayImage");
imgObject.onclick = swapImage;
```

Но будьте осторожны, чтобы не сделать эту ошибку:

```
imgObject.onclick = swapImage();
```

Этот код выполняет функцию `swapImage()` и использует ее результат (если функция возвращает таковой), чтобы установить обработчик события. Это почти наверняка не то, что нам нужно.

Чтобы понять, что в действительности происходит при щелчке по элементу `<img>`, нужно разобраться с кодом функции `swapImage()`. Эта функция берет элемент `<img>` и модифицирует его атрибут `src`, чтобы тот указывал на новое изображение (рис. П2.3):

```
// Отслеживаем состояние изображения — день или ночь.
var dayTime = true;

// Эта функция выполняется, когда происходит событие onClick.
function swapImage() {
  var imgObject = document.getElementById("dayImage");

  // Меняем день на ночь или наоборот и обновляем изображение
  // в соответствии с новым состоянием.
  if (dayTime == true)  { dayTime = false;
                          imgObject.src = "cloudy.jpg"; }
  else { dayTime = true;
         imgObject.src = "sunny.jpg"; }
}
```

Иногда событие передает информацию своему обработчику. Чтобы получить эту информацию, функции нужно дать один параметр. По соглашению, этот параметр обычно называется `event` или просто `e`:

```
function swapImage(e)  {
  ...
}
```

Свойства объекта события зависят от этого события. Например, событие `onMouseMove` предоставляет текущие координаты указателя мыши (которые используются в программе рисования в *разд. "Создание простой программы рисования" главы 6*).

Нужно отметить еще одну деталь. При подключении события название события *в обязательном порядке* пишется строчными буквами. Это отличается от подклю-

чения события с использованием атрибута в HTML. В отличие от HTML JavaScript обращает внимание на регистр.

Примечание

В книге названия событий даются в соответствии с соглашением CamelCase[1], по которому каждое новое слово в названии пишется с прописной буквы, например `onLoad` или `onMouseOver`. Но в коде, согласно требованиям JavaScript, все названия пишутся ***строчными буквами***, т. е. `onload` или `onmouseover`.

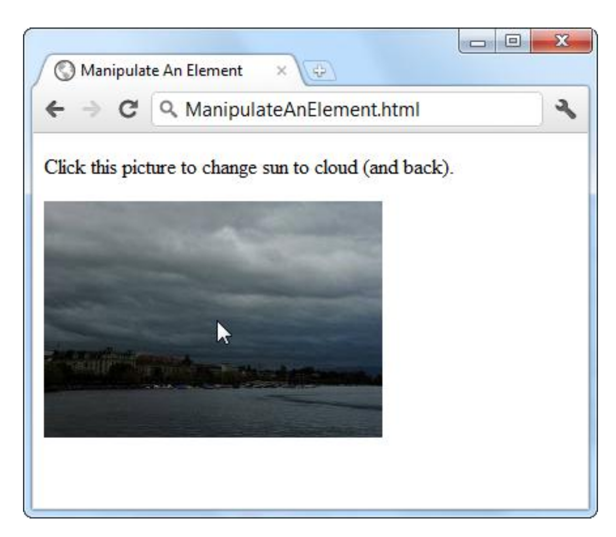


Рис. П2.3. Щелчок по изображению активирует событие `onClick` этого изображения. Это событие в свою очередь активирует функцию, которая загружает новое изображение

Подставляемые в строку функции

Для работы предыдущего примера функцию `swapImage()` нужно определить где-то в другом месте кода. Иногда вместо этого функцию можно определить в том же месте кода, где она подключается к событию. Такая функция называется *подставляемой в строку* (inline function).

Вот пример подключения подставляемой в строку функции к событию `onClick`:

```
var imgObject = document.getElementById("dayImage");
imgObject.onclick = function() {
  // Код функции swapImage() теперь вставляется сюда.
  if (dayTime == true) {
    dayTime = false;
    imgObject.src = "cloudy.jpg";
  }
```

[1] См. **http://ru.wikipedia.org/wiki/Camel_case**. — *Прим. пер.*

```
  else {
    dayTime == true;
    imgObject.src = "sunny.jpg";
  }
};
```

Этот подход к обработке событий применяется менее часто, чем использование отдельной, именованной функции. Но это все равно полезный метод для определенных целей и иногда используется в примерах в этой книге.

ПРИМЕЧАНИЕ

Подставляемые в строку функции иногда полезны при исполнении асинхронных задач, т. е. задач, которые исполняются браузером в фоновом режиме. По завершению выполнения асинхронной задачи браузер активирует событие, чтобы известить код об этом. Иногда самым легким подходом в этой ситуации будет вставка кода для обработки *завершения* задачи непосредственно рядом с кодом, который активировал выполнение этой задачи. (В конце *разд. "Вставка в холст изображений" главы 7* представлен пример такого кода, который асинхронно загружает изображение, а потом обрабатывает его.)

Наконец, есть еще один тип подставляемой в строку функции, которая используется во многих примерах в этой книге. Это обработчик события `onLoad` окна, который активируется после полной загрузки и отображения страницы. Это будет логичной точкой для запуска кода. Если попытаться исполнить код до полной загрузки страницы, то возможно возникновение проблем, если объект для требуемого элемента еще не был создан. Далее показан пример такой функции:

```
<script>
  window.onload  = function() {
    alert("The page has just finished loading.");
  }
</script>
```

Этот подход позволяет не беспокоиться о местоположении блока сценариев. Таким образом, код инициализации можно разместить там, где ему место — в блоке `<head>` вместе с остальными функциями JavaScript.

Предметный указатель

A

C

E

F

G

H

J

L

M

P

R

U

V



W



X

Y

Z

А

Б

В

Г

Д

З

И

Ч

Ш

Э

Я